Compiling Embedded Languages

Conal Elliott[1], Sigbjørn Finne[2] and Oege de Moor[3]

[1] Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
[2] Galois Connections, Inc.
3875 SW Hall Blvd.
Beaverton, OR 97005, USA
[3] Oxford University Computing Laboratory,
Wolfson Building, Parks Road,
Oxford, OX1 3QD, England

Abstract. Functional languages are particularly well-suited to the implementation of interpreters for domain-specific embedded languages (DSELs). We describe an implemented technique for producing *optimizing compilers* for DSELs, based on Kamin's idea of DSELs for program generation. The technique uses a data type of syntax for basic types, a set of smart constructors that perform rewriting over those types, some code motion transformations, and a back-end code generator. Domain-specific optimization results from chains of rewrites on basic types. New DSELs are defined directly in terms of the basic syntactic types, plus host language functions and tuples. This definition style makes compilers easy to write and, in fact, almost identical to the simplest embedded interpreters. We illustrate this technique with a language *Pan* for the computationally intensive domain of image synthesis and manipulation. [1]

1 Introduction

The "embedded" approach has proved an excellent technique for specifying and prototyping domain-specific languages (DSLs) [14]. The essential idea is to augment a "host" programming language with a domain-specific library. Modern functional host languages are flexible enough that the resulting combination has more the feel of a new language than a library. Most of the work required to design, implement and document a language is inherited from the host language. Often, performance is either relatively unimportant, or is adequate because the domain primitives encapsulate large blocks of work. When speed is of the essence, however, the embedded approach is problematic. It tends to yield inefficient *interpretive* implementations. Worse, these interpreters tend to perform redundant computation.

[1] For submission to a special issue of the *Journal of Functional Programming* on Semantics, Applications, and Implementation of Program Generation (SAIG), and based on a paper that appeared at the *SAIG 2000* workshop as part of PLI 2000.

We have implemented a language *Pan* for image synthesis and manipulation, a computationally demanding problem domain. A straightforward embedded implementation would not perform well enough, but we did not want to incur the expense of introducing an entirely new language. Our solution is to embed an *optimizing compiler* rather than an interpreter. Embedding a compiler requires some techniques not normally needed in embedded language implementations, and we report on these techniques here. Pleasantly, we have been able to retain a simple programming interface, almost unaffected by the compiled nature of the implementation. The generated code runs very fast, and there is still much room for improvement.

Our compiler consists of a relatively small set of domain definitions, on top of a larger domain-independent framework. The framework may be adapted for compiling other DSLs, and handles (a) optimization of expressions over numbers and Booleans, (b) code motion, and (c) code generation. A new DSL is specified and implemented by defining the key domain types and operations in terms of the primitive types provided by the framework and host language. Moreover, these definitions are almost identical to what one would write for a very simple interpretive DSL implementation.

Although a user of our embedded language writes in Haskell, we do not have to parse, type-check, or compile Haskell programs. Instead, the user *runs* his/her Haskell program to produce an optimized program in a simple target language that is first-order, call-by-value, and mostly functional. (Thus the user's program is a "generating extension" in the terminology of partial evaluation [13].) Generated target language programs are then given to a simple compiler (also implemented in Haskell) for code motion and back-end code generation. In this way, the host language (Haskell here) acts as a powerful macro (or *program generator*) language, but is completely out of the picture at run-time. Unlike most macro languages, however, Haskell is statically-typed and higher order, and is more expressive and convenient than the underlying target language.

Because of this embedded compiler approach, integration of the DSEL with the host language (Haskell) is not quite as fluid and general as in conventionally implemented DSELs. Some host language features, like lists, recursion, and higher-order functions are not available to the final executing program. These features may be used in source programs, but disappear during the compilation process. For some application areas, this strict separation of features between a full-featured compilation language and a less rich runtime language may be undesirable, but in our domain, at least, it appears to be perfectly acceptable. In fact, we typically write programs without being conscious of the difference.

The contributions of this paper are as follows:

- We present a general technique for implementing *embedded optimizing* compilers, extending Kamin's approach [17] with algebraic manipulation.
- We identify a key problem with the approach, efficient handling of sharing, and present techniques to solve it (bottom-up optimization and common subexpression elimination).

- We illustrate the application of our technique to a demanding problem domain, namely image synthesis and manipulation.
- We identify a number of desirable features for simplifying experimentation with global optimizations (such as code motion).

While this paper mainly discusses embedded language compilation, a companion paper goes into more detail for the Pan image synthesis language [8]. That paper contains many more visual examples, as does [6].

2 Language embedding

The embedding approach to DSL construction goes back at least to Landin's famous "next 700" paper [19]. The essential idea is to use a single existing "host" programming language that provides useful generic infrastructure (grammar, scoping, typing, function- and data-abstraction, etc), and augment it with a domain-specific vocabulary consisting of one or more data types and functions over those types. Thus the design, implementation, and documentation work required for a new "language" is kept to a minimum, while the result has plenty of room to grow. These merits and some drawbacks are discussed, e.g., in [7, 14].

One particularly elegant realization of the embedding idea is the use of a modern functional programming language such as ML or Haskell as the host. In this setting, the domain-specific portions can sometimes be implemented as a simple denotational semantics, as suggested by Kamin and Hyatt [18, Section 3]. For example, consider the problem domain of image synthesis and manipulation. A simple semantics for images is function from continuous 2D space to colors. The representation of colors includes blue, green, red, and opacity ("alpha") components:

$$\begin{aligned}&\textbf{type}\ \mathit{Image} = \mathit{Point} \to \mathit{Color}\\ &\textbf{type}\ \mathit{Point} = (\mathit{Float}, \mathit{Float})\\ &\textbf{type}\ \mathit{Color} = (\mathit{Float}, \mathit{Float}, \mathit{Float}, \mathit{Float})\end{aligned}$$

It is easy to implement operations like image overlay (with partial opacity), assuming a corresponding function, *cOver*, on color values:

$$a\ `\mathit{over}`\ b = \lambda p \to a\ p\ `\mathit{cOver}`\ b\ p$$

Another useful type is spatial transformation, which may be defined simply as a mapping from 2D space to itself:

$$\textbf{type}\ \mathit{Transform} = \mathit{Point} \to \mathit{Point}$$

This model makes it easy to define some familiar transformations:

$$\begin{aligned}&\mathit{translate}\ (dx,\ dy) = \lambda (x, y) \to (x + dx,\ y + dy)\\ &\mathit{scale}\ (sx,\ sy) = \lambda (x, y) \to (sx * x,\ sy * y)\\ &\mathit{rotate}\ \mathit{ang} = \lambda (x, y) \to (x * c - y * s,\ y * c + x * s)\end{aligned}$$

```
  where
    c = cos ang
    s = sin ang
```

While these definitions can be directly executed as Haskell programs, performance is not good enough for practical use. Our first attempt to cope with this problem was to use the Glasgow Haskell compiler's facility for stating transformations as rewrite rules in source code [23]. Unfortunately, we found that the interaction of such rewrite rules with the general optimizer is hard to predict: in particular, we often wish to inline function definitions that would normally not have been inlined. Furthermore, there are a number of transformations (if-floating, certain array optimizations) that are not easy to state as rewrite rules. We therefore abandoned use of the Haskell compiler, and decided to build a dedicated compiler instead. We will discuss this decision further in Section 11.

3 Embedding a compiler

In spite of our choice to implement a dedicated compiler, we would like to retain most of the benefits of the embedded approach. We resolve this dilemma by applying Kamin's idea of DSELs for program generation [17] (which is a form of *multi-stage programming* [22]). That is, replace the *values* in our representations by *program fragments* that represent these values. While Kamin used strings to represent program fragments, algebraic data types greatly facilitate our goal of compile-time optimization. For instance, an expression type for *Float* would contain literals, arithmetic operators, and other primitive functions that return *Float*.

```
data FloatE =
   LitFloat Float
  |AddF FloatE FloatE | MulF FloatE FloatE | ...
  |Sin FloatE | Sqrt FloatE | ...
```

We can define expression types *IntE* and *BoolE* similarly.

What about tuples and functions? Following Kamin, we simply adopt the host language's tuple and functions, rather than creating new syntactic representations for them. Since optimization requires inspection, representing functions as functions poses a problem. The solution we use is to extend the base types to support "variables". Then to inspect a function, apply it to a new variable (or tuple of variables as needed), and look at the result.

```
data FloatE = ... | VarFloat String    — named variable
```

These observations lead to a hybrid representation. Our *Image* type will still be represented as a function, but over *syntactic* points, rather than actual ones. Moreover, these syntactic points are represented not as expressions over number pairs, but rather as pairs of expressions over numbers. Similarly for colors. Thus:

$$\begin{aligned}
&\textbf{type } \mathit{ImageE} &&= \mathit{PointE} \to \mathit{ColorE}\\
&\textbf{type } \mathit{TransformE} &&= \mathit{PointE} \to \mathit{PointE}\\
&\textbf{type } \mathit{PointE} &&= (\mathit{FloatE}, \mathit{FloatE})\\
&\textbf{type } \mathit{ColorE} &&= (\mathit{FloatE}, \mathit{FloatE}, \mathit{FloatE}, \mathit{FloatE})
\end{aligned}$$

The definitions of operations over these types can often be made identical to the ones for the non-expression representation, thanks to overloading. For instance *translate*, *scale*, and *rotate* have precisely the definitions given in Section 2 above. The *meaning* of these definitions, however, is quite different. The arithmetic operators and the functions *cos*, *sin* as well as several others have been overloaded. The *over* function is also defined exactly as before. Only the types *BoolE*, *IntE*, and *FloatE* of expressions over the usual "scalar" value types *Bool*, *Int*, and *Float*, are represented as expressions, using constructors for their primitive operations. Assuming that these base types are adequate, a DSL is just as easy to define and extend as with a simple, non-optimizing embedded interpreter. Otherwise new syntactic types and/or primitive operators may be added.

As an example of how the hybrid technique works in practice, consider rotating by an angle of $\pi/2$. Using the definition of *rotate* plus a bit of simplification on number expressions (*FloatE*), the compiler simplifies *rotate* $(\pi/2)\,(x, y)$ to $(-y, x)$.

Admittedly, the picture might not always be this rosy. For instance, some properties of high-level types require clever or inductive proofs. Formulating these properties as high-level rules would eliminate the need for a generic compiler to rediscover them. So far this has not been a problem for our image manipulation language, but we expect that for more substantial applications, it may be necessary to layer the compilation into a number of distinct abstract levels. In higher levels, domain types and operators like *Image* and *over* would be treated as opaque and rewritten according to domain-specific rules, while in lower levels, they would be seen as defined and expanded in terms of simpler types like *Point* and *Color*. Those simpler types would themselves be expanded at lower levels of abstraction.

4 Inlining and the sharing problem

The style of embedding described above has the effect of *inlining* all definitions, and β-reducing resulting function applications, before simplification. This inlining is beneficial in that it creates many opportunities for rewriting. A resulting problem, however, is that uncontrolled inlining often causes a great deal of code replication. To appreciate this problem, consider the following example spatial transform. It rotates each point about the origin, through an angle proportional to the point's distance from the origin. The parameter r is the distance at which an entire revolution (2π radians) is made.

$$\begin{aligned}
&\mathit{swirling} :: \mathit{FloatE} \to \mathit{TransformE}\\
&\mathit{swirling}\ r\ =\ \lambda\, p\ \to\ \mathit{rotate}\ (\mathit{distO}\ p\ *\ (2\,\pi\ /\ r))\ p
\end{aligned}$$

$$
\begin{aligned}
&distO :: PointE \to FloatE \\
&distO\,(x, y) = sqrt\,(x * x + y * y)
\end{aligned}
$$

Evaluating $swirling\ r\ (x, y)$ yields an expression with much redundancy.

$$
\begin{aligned}
&(x * cos\,(sqrt\,(x * x + y * y) * 2\,\pi\,/\,r) \\
&\quad - y * sin\,(sqrt\,(x * x + y * y) * 2\,\pi\,/\,r) \\
&,y * cos\,(sqrt\,(x * x + y * y) * 2\,\pi\,/\,r) \\
&\quad + x * sin\,(sqrt\,(x * x + y * y) * 2\,\pi\,/\,r)\,)
\end{aligned}
$$

The problem here is that *rotate* uses its argument four times (twice via each of *cos* and *sin*) in constructing its results. Thus expressions passed to *rotate* get replicated in the output. In our experience with image synthesis, the trees resulting from inlining and simplification tend to be enormous, compared to their underlying representation as graphs. If *swirling r* were composed with *scale* (u, v) before being applied to (x, y), the two multiplications due to *scale* would each be appear twice in the argument to *sqrt*, and hence eight times in the final result. Note that this problem of redundancy is a consequence of our choice to use an *embedded* language, in which the language's "let", lambda, and application are simply those of the host language (Haskell). This choice prevents us from using a non-inlined representation.

In an interpretive implementation, we would have to take care not to evaluate shared expressions redundantly. Memoization is a reasonable way to avoid such redundance. For a compiler, memoization is not adequate, because it must produce an external representation that captures the sharing. What we really want is to generate local definitions when helpful. To produce these local definitions, our compiler performs common subexpression elimination (CSE), as described in Section 8.

5 Static typing

Should there be one expression data type per value type (*Int*, *Float*, *Bool*, etc) as suggested above, or one for all value types? Separate expression types make the implementation more statically-typed, and thus prevent many bugs in implementation and use. Unfortunately, they also lead to redundance for variables, binding, and polymorphically and overloaded expression operators (e.g., if-then-else and addition, respectively), as well as polymorphic compiler-internal operations on terms (e.g., substitution and CSE).

Instead, we use a single all-encompassing expression data type *DExp* of "dynamically-typed expressions":

```
data DExp =
    LitInt Int | LitFloat Float | LitBool Bool
  | Var Id Type | Let Id Type DExp | If DExp DExp DExp
  | Add DExp DExp | Mul DExp DExp | ...
  | Sin DExp | Sqrt DExp | ...
  | Or DExp DExp | And DExp DExp | Not DExp | ...
```

It is unfortunate that the choice of a single *DExp* type means that one cannot simply add another module containing a new primitive type and its constructors and rewrite rules. For now we are willing to accept this limitation, but future work may suggest improvements.

The *DExp* representation removes redundance from representation and supporting code, but loses type safety. To combine advantages of both approaches, we augment the dynamically-typed representation with the technique of "phantom types" [20]. The idea is to define a type constructor (*Exp* below) whose parameter is not used, and then to restrict types of some functions to applications of the type constructor. For convenience, define abbreviations for the three supported base types as well:

```
data Exp α  =  E DExp

type BoolE  =  Exp Bool
type IntE   =  Exp Int
type FloatE =  Exp Float
```

For static typing, it is vital that *Exp* α be a new type, rather than just a type synonym of *DExp*.

Statically-typed functions are conveniently defined via the following functionals, where typ_n turns an n-ary *DExp* function into an n-ary *Exp* function.

```
typ1 ::(DExp → DExp) → (Exp a → Exp b)
typ2 ::(DExp → DExp → DExp) → (Exp a → Exp b → Exp c)

typ1 f (E e1)         =  E (f e1)
typ2 f (E e1) (E e2)  =  E (f e1 e2)
```

and so on for typ_3, typ_4, *etc.* The type-safe friendly names +, ∗, *etc.*, come from applications of these static typing functionals in type class instances:

```
instance Num IntE
  where
    (+)         =  typ2 Add
    (*)         =  typ2 Mul
    negate      =  typ1 Negate
    fromInteger =  E . LitInt . fromInteger
```

Type constraints inherited from the *Num* class ensure that the newly defined functions be applied only to *Int* expressions and result in *Int* expressions. For instance, here

```
(+) :: IntE → IntE → IntE
```

The important point here is that we do not rely on type inference, which would deduce too general a type for functions like "+" on *Exp* values. Instead we state restricted type signatures.

Other definitions provide a convenient and type-safe primitive vocabulary for *FloatE*. Unfortunately, the *Bool* type is wired into the signatures of operations like $\geq$ and $||$. Pan therefore provides alternative names ending in a distinguished character, which is "*E*" for alphanumeric names (e.g., "*notE*") and "∗" for non-alphanumeric names (e.g., "<∗").

6 Algebraic optimization and smart constructors

An early Pan implementation was based on the Mag program transformation system [5]. Generation in this implementation was much too slow, mainly because Mag redundantly rewrote shared subterms. To avoid this problem, we now do all optimization *bottom-up*, as part of the construction of expressions. Then the host language's evaluate-once operational semantics prevents redundant optimization. Non-optimized expressions are never constructed. The main drawback is that optimization is context-free. (An optimization can, however, delve arbitrarily far into an argument term.)

Optimization is packaged up in "smart constructors", each of which accomplishes the following:

- constant-folding;
- if-floating;
- constructor-specific rewrites such as identities and cancellation rules;
- data type constructor application when no optimizations apply; and
- providing a statically-typed interface.

As an example, Figure 1 shows a smart constructor for conjunction over expressions. In fact, because all smart constructors perform constant folding and if-floating, the real definition is more factored, but it does the same work. Smart constructors must be programmed with care, because they work on the dynamically-typed representation, and hence cannot be verified by the host language's type checker.

Because Haskell's if-then-else is not overloadable, Pan uses *ifE* for syntactic conditionals, based on an underlying dynamically-typed *ifD*.

$$ifD :: DExp \rightarrow DExp \rightarrow DExp \rightarrow DExp$$

$$\begin{aligned}
ifD\ (LitBool\ True\ )\ a\ b &= a \\
ifD\ (LitBool\ False)\ a\ b &= b \\
ifD\ (Not\ c)\ a\ b &= ifD\ c\ b\ a \\
ifD\ (If\ c\ d\ e)\ a\ b &= ifD\ c\ (ifD\ d\ a\ b)\ (ifD\ e\ a\ b) \\
ifD\ c\ a\ b &= ifZ\ c\ a\ b
\end{aligned}$$

The function *ifZ* simplifies redundant or impossible conditions.

The statically-typed *ifE* function is overloaded.

$$\textbf{class}\ Syntactic\ a\ \textbf{where}\ ifE :: BoolE \rightarrow a \rightarrow a \rightarrow a$$

```
— Type-safe smart constructor
(&&*) :: BoolE → BoolE → BoolE
(&&*) =typ2 andD
— Non-type-safe smart constructor
andD :: DExp → DExp → DExp
— Constant folding
andD (LitBool a) (LitBool b) = LitBool (a && b)
— If-floating
andD (If c a b) e2 =ifD c (andD a e2) (andD b e2)
andD e1 (If c a b) =ifD c (andD e1 a) (andD e1 b)
— Cancellation rules
andD e (LitBool False) =false
andD (LitBool False) e =false
andD e (LitBool True) =e
andD (LitBool True) e =e
— Others
andD (Not e) (Not e')      =notE (e || * e')
andD e e' | e == e'        =e
andD e e' | e == notE e' =false
— Finally, the data type constructor
andD e e' = And e e'
```

Fig. 1. Simplification rules for conjunction

instance *Syntactic* (*Exp a*) **where** *ifE* = typ_3 *ifD*

Other overloadings include functions and tuples. In the latter case, conditions are pushed downward. Later when the resulting tuple is consumed to form a single (scalar-valued) expression, if-floating typically causes the redundant conditions to float, to form a cascade of redundant conditionals, which are coalesced by *ifZ*.

As an example of if-floating, consider the following example (given in familiar concrete syntax, for clarity):

$$sin\ ((\textbf{if}\ x < 0\ \textbf{then}\ 0\ \textbf{else}\ x)\ /\ 2)$$

If-floating without simplification would yield

$$\textbf{if}\ x < 0\ \textbf{then}\ sin(0/2)\ \textbf{else}\ sin(a/2)$$

Replacement followed by two constant foldings ($0/2$ and $sin\ 0$) results in

$$\textbf{if}\ x < 0\ \textbf{then}\ 0\ \textbf{else}\ sin(a/2)$$

If-floating causes code replication, sometimes a great deal of it. CSE factors out the "first-order" replication, i.e., multiple occurrences of expressions, as with e_2 for the first if-floating clause in Figure 1. There is also a *second-order* replication going on, as seen above before simplification. The context $sin\ (\bullet / 2)$ appears

twice. Fortunately for this example, one instance of this context simplifies to 0. In other cases, there may be little or no simplification. We will return to this issue in Section 11.

We should stress at this point that we intend the algebraic optimizations to be *refinements*: upon evaluation, the optimized version of an expression e should yield the same value as e *whenever evaluation of* e *terminates*. It is possible, however, for simplified version to yield a well-defined result when e does not. This could happen for example when a boolean expression $e \mathbin{\&\&} * \mathit{false}$ would raise a division-by-zero exception, while the simplified version would instead evaluate to *false*.

7 Adding context

More optimization becomes possible when the usage context of a DSL computation becomes visible to the compiler. For instance, after composing an image, a user generally wants to display it in a window. The representation of images as $\mathit{PointE} \to \mathit{ColorE}$ suggests iteratively sampling at a finite grid of pixel locations, converting each pixel color to an integer for the display device.[2] Our first Pan compiler implementation took this approach, that is it generated machine code for a function that maps a pixel location to a 32-bit color encoding. While this version was much faster than an interpretive implementation, its efficiency was not satisfactory. For one thing, it requires a function call per pixel. More seriously, it prevent any optimization across several pixels or rows of pixels.

To address the shortcomings of the first compiler, we made visible to the optimizer the two-dimensional iteration that samples and stores pixel values. In fact, to get more use out of compilation, we decided to compile the display of not simply static images, but animations, represented as functions from time to image. (We go even further, generating code for nearly arbitrarily parameterized images, with automatic generation of user interfaces for the run-time parameters.)

The main function *display*, defined in Figure 2, converts an animation into a "display function" that is to be invoked just once per frame. (Recall that *FloatE* and *IntE* are the "syntactic" base type short-hands for *Exp Float* and *Exp Int*, respectively.) A display function consumes a time, viewing transform (zoom factor and XY pan), window size, and a pointer to an output pixel array. It is the job of the viewer to come up with all these parameters and pass them into the display function code.

The critical point here is that (a) the *display* function is expressed in the embedded language, and (b) *display* is applied to its *anim* parameter (of type $\mathit{TimeE} \to \mathit{Image}$) *at compile time*. This compile-time application allows the

[2] For a faithful presentation, the Pan viewer performs anti-aliasing by making several display passes, each with a random, sub-pixel offset, and averages the results together. The display window is repainted after each pass, so the appearance improves gradually over time. For efficiency, the generated code actually modifies the output bitmap in place.

```
type TimeE = FloatE
type Anim = TimeE → ImageE
type DisplayFun = TimeE → VTrans → VSize → IntE → ActionE
type VSize = (IntE, IntE)   — view size: width & height in pixels
type VTrans = (FloatE, FloatE, FloatE)   — view transform: pan XY, zoom

display :: Anim → DisplayFun
display anim = λ t (panX, panY, zoom) (w, h) output →
  loop h (λ j →
    loop w (λ i →
      setInt (output + 4 * (j * w + i)) (
        toBGR24 (
          anim t (
            zoom * i2f (i − w`div` 2) + panX,
            zoom * i2f (j − h`div` 2) + panY )))))
```

Fig. 2. Animation display function

code for *display* and *anim* to be combined and optimized, and lets some computations be moved outside of the inner or outer loop. (In fact, our compiler goes further, allowing focused recomputations when only some display parameters change, thanks to a simple dependency analysis.)

The *ActionE* type represents an action that yields no value, much like Haskell's type *IO* (). It is supported by a small number of *DExp* constructors and corresponding statically-typed, optimizing wrapper functions. The first takes an address (represented as an integer) and an integer value, and it performs the corresponding assignment. The second is like a for-loop. It takes an upper bound, and a loop body that is a function from the loop variable to an action. The loop body is executed for every value from zero up to (but not including) the upper bound.

```
setInt :: IntE → IntE → ActionE
loop   :: IntE → (IntE → ActionE) → ActionE
```

According to *display*, a generated display function will loop over Y and X, and set the appropriate member of its output array to a 32-bit (thus multiplication by four) color value. Aside from calculating the destination memory address, the inner loop body samples the animation at the given time and position. The spatial sampling point is computed from the loop indices by placing the image's origin in the center of the window (thus the subtraction of half the window width or height) and then applying the user-specified dynamic zoom and pan (using *i2f* for int-to-float conversion). In fact, the optimized code is much more efficient, thanks to code motion techniques described in Section 8 and illustrated in Appendix A.

8 Code motion

Once all the above optimizations have been applied, the result is a directed acyclic graph. That graph represents a rather large expression tree, as explained in Section 4. Nodes in the graph with more than one parent represent expressions that were replicated during the inlining and rewriting process. We wish to take this graph, and make the sharing structure explicit using let-bindings: we can then apply a number of global optimizations (in particular code motion) that are difficult to phrase as simple algebraic identities.

8.1 Converting dags to lets

The first problem is to make the internal graph structure of a value of type *DExp* explicit, which we do in two steps. The first step converts the expression to a graph with a designated node, and the second turns the graph back into an expression, introducing *let* bindings for shared subexpressions.

$$expToDag :: DExp \rightarrow (Graph, Node)$$
$$dagToLet :: (Graph, Node) \rightarrow DExp$$
$$share :: DExp \rightarrow DExp$$
$$share = dagToLet \circ expToDag$$

Unfortunately, implementing the *expToDag* transformation in Haskell requires using non-declarative pointer manipulation. It might be possible to apply the work of Sands and Claessen to avoid this ugly departure from a declarative implementation, or at least make it as innocuous as possible [3]. They extend Haskell with reference types, and show that many compiler transformations remain valid. Reference types are precisely what is needed to capture the notion of sharing.

The *dagToLet* function relies on extending the expression data type with variable binding:

$$data\ DExp = \ldots \mid Let\ Id\ DExp\ DExp$$

$$type\ Id = String$$

$$data\ Type = Bool \mid Int \mid Float \mid \ldots$$

Since the variable references (*Var*) and bindings (*Let*) are only introduced through *dagToLet* and other transformations, the programmer cannot create expressions with references to unbound variables.

We require that for all e, the result of $dagToLet\ (expToDag\ e)$ is an expression that is equivalent to e under the semantics of our embedded language. In particular, the result of the conversion should have the same (or better) termination behaviour. As we shall see, fulfilling that requirement is complicated by the fact that *Let* has strict semantics.

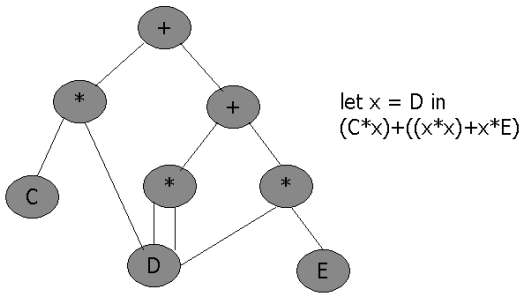

Fig. 3. A dag and the corresponding let-expression.

Each node that has more than one parent and at least one child represents a non-trivial shared subexpression, and could potentially become the body of a *Let*. The main problem is to decide where such a *Let* should be placed. As an example, consider the graph in Figure 3. These and similar examples might lead us to believe that the appropriate placement for a let-expression is at the lowest common ancestor of all occurrences of its body.

Unfortunately that would be incorrect, as shown by the example in Figure 4. In the original expression, if both the tests A and B fail, it is not necessary to evaluate D. Under a strict interpretation of the let-expression, D does get evaluated, and might cause a runtime error. This motivates the following definition for the placement of let-abstractions. A let-abstraction for a shared subexpression D is placed at each ancestor node m such that

- At least two children of m contain D, and
- if the expression rooted at m is evaluated, so is at least one of the occurrences of D below it, and
- no ancestor of m has the above two properties.

In particular, a single expression may be abstracted in more than one *Let*, and it may also happen that a shared subexpression is not abstracted at all. One could choose to ignore the second clause for shared subexpressions D that cannot diverge in the evaluation of m. It only pays to abstract such expressions D if they are very large, or costly to evaluate. If the expression does not contain any conditional expressions, the above definition says that a single let-abstraction is placed at the lowest common ancestor of its body's occurrences.

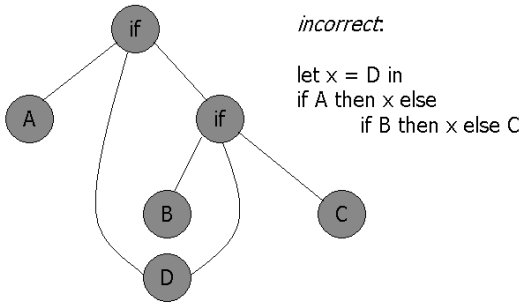

Fig. 4. A dag and an incorrect let-expression.

Assume there are n nodes in the original graph. There exist some ingenious algorithms for finding the lowest common ancestors of all repeated subexpressions in time $\mathcal{O}(n \cdot \alpha(n))$ (where α is the inverse of the Ackermann function) [11, 1], or even $\mathcal{O}(n)$ [10]. We are considering how these algorithms may be adapted to solve the above problem efficiently — the current implementation uses a naive algorithm whose worst-case performance is quadratic.

Some readers may wonder how we can end up with shared subexpressions that cannot be abstracted to a common *Let*. Does it mean that some of our algebraic optimizations, as discussed in previous sections, are unsound? Certainly not. It is simply a consequence of the fact that the sharing in our graph representation is syntactic, and has no semantic meaning. To illustrate, consider the simplification of

$$\textbf{if } c \textbf{ then } a \textbf{ else } b + \textbf{if } c' \textbf{ then } a' \textbf{ else } b'$$

(we write if-then-else in lieu of *ifD* for clarity). The if-floating transformation discussed in Section 6 will result in

```
if c then
  if c'
  then a + a'
  else a + b'
else
  if c'
  then b + a'
  else b + b'
```

This transformed expression is equivalent to the original, but we cannot safely share a' and b' by introducing *Let*s, even though a' and b' are shared in the graph representation. We can, however, safely extract c', because it appears twice in strict positions.

8.2 Common subexpression elimination

Once we have the let-representation of the expression that resulted from rewriting, we can apply other optimizations. In particular, we apply ordinary common subexpression elimination, again taking care not to move expressions out of the branches of a conditional unless it is safe to do so. It may seem superfluous to do CSE after the conversion from dags to lets, but we have found that there are a fair number of repeated subexpressions that do not result from inlining.

Our implementation of common subexpression elimination uses the standard hashing technique, implemented via an attribute grammar [15]. By writing this transformation as an attribute grammar, we can apply it simultaneously with the optimization of array expressions, discussed in the next subsection.

8.3 Optimizing array expressions

Our final set of global transformations improve loops and array expressions. Here we shall only discuss array expressions, which are the basis of our treatment of bitmaps. Loops (as discussed in Section 7) are treated in a very similar fashion.

The type of expressions contains two constructors related to arrays: one named *mkArray* to construct new arrays, and *readArr* to read the value of an array at a given index:

```
data DExp =
  ...
  |MkArray Id DExp DExp | ReadArr DExp DExp
```

In the constructor application $MkArray\ x\ size\ body_x$, the first argument is an identifier x, which is a bound index variable that ranges from 0 up to (but not including) the integer expression *size*. The $body_x$ argument gives the value that should be stored at index x in the newly created array.

As always, the statically typed smart constructor hides identifier introduction:

```
mkArray :: IntE → (IntE → Exp a) → ArrayE a
```

The typed version of *ReadArr* ensures consistency:

```
readArr :: ArrayE a → IntE → Exp a
```

The semantics of *mkArray* and *readArr* (and their underlying constructors) are as expected:

$$readArr\ (mkArray\ size\ f)\ i = f\ i \quad \text{— if } 0 \leq i < size$$

Our first optimization corresponds to the hoisting of loop-invariant code in traditional compilers (see e.g. [2]). If a subexpression of the body of an array construction does not depend on the index variable, it can be precomputed:[3]

$$\begin{array}{l} \quad mkarray\ size\ (\lambda\, i\ \rightarrow\ A[sub]) \\ = \\ \quad \textbf{let}\ x\ =\ sub\ \textbf{in}\ mkArray\ size\ (\lambda\, i\ \rightarrow\ A[x]) \end{array}$$

subject to several side conditions:

- The variable i does not occur freely in sub.
- The $size$ of the array must be greater than 0.
- Every evaluation of the expression matching $A[sub]$ involves an evaluation of sub.
- The expression sub is not trivially small.
- The expression $A[x]$ does not contain sub as a subexpression (so we are abstracting all occurrences of sub).
- The subexpression sub is not contained in another subexpression of $A[sub]$ that satisfies the above conditions.

Note the similarity to the conditions we employed in converting dags to lets. Again it is helpful to relax the second and third restrictions, for expressions sub that are expensive to evaluate and yet are guaranteed to be total.

Another important optimization of array expressions occurs when there are two nested array constructions, and a subexpression only depends on the inner index variable. In such cases, one might precompute the subexpression for each value of the inner loop variable, and store the results of the precomputation in an array.

$$\begin{array}{l} \quad mkArray\ size_1\ (\lambda\, i\ \rightarrow\ A[mkArray\ size_2\ (\lambda\, j\ \rightarrow\ B[sub])]) \\ = \\ \quad \textbf{let}\ x\ =\ mkArray\ size_2\ (\lambda\, j\ \rightarrow\ sub)\ \textbf{in} \\ \qquad mkArray\ size_1\ (\lambda\, i\ \rightarrow\ A[mkArray\ size_2\ (\lambda\, j\ \rightarrow\ B[readArr\ x\ j])]) \end{array}$$

Again there are several side conditions:

- The variable i does not occur freely in sub.
- The array sizes $size_1$ and $size_2$ are greater than 0.
- Every evaluation of $B[sub]$ involves an evaluation of sub.
- The subexpression sub is not trivially small.
- The expression $B[y]$ does not contain sub as a subexpression.
- The subexpression sub is not contained in another expression that satisfies the above conditions.

[3] The pattern $A[e]$ matches an expression containing one or more occurrences of a subexpression e, with A bound to an "expression with holes".

Both of the above optimizations on arrays, and common subexpression elimination are implemented through a single attribute grammar. Although the attribute grammar uses the above transformations to lift subexpressions of arrays as far as possible in one pass, it is not idempotent because the movement may introduce new common subexpressions. In experiments it appears that two or three iterations suffice to reach a fixpoint.

Nested loops are optimized in exactly the same way, which is particularly useful with the *display* function in Figure 2. See Appendix A for an example.

9 Code generation

Having performed code motion and loop hoisting, we are in good shape to start generating some code. The output of the code motion pass could either be interpreted or compiled, but we choose to compile.

In the spirit of embedding in a functional language, the final *DExp* tree could be translated to a Haskell or ML function definition, relying on an existing optimizing compiler to do code generation. In the case of image processing and Pan we chose not to do this, mainly for performance reasons and the wish to exploit features of our target platform. However, for other DSLs, generating a Haskell/ML function might be the most sensible route.

The *DExp* can be converted into either a C function or native code. Going via C is reasonably straightforward, but requires a little bit of care in places. For instance, we need to account for the fact that C doesn't really have expression level variable binding support. Also, since the *MkArray* construct requires the allocation of an array followed by a loop that fills it in, i.e., it maps to a sequence of C statements, so it cannot be embedded within a C expression. Consequently, we must massage the *DExp* tree before translating it to valid C code.

In the case of Pan, the generated code is then compiled by a C compiler and linked into a viewer that displays the specified image effect.

Going via C allows us to reuse the strengths of a C code generator, but most C compilers do not generate code that targets various instruction set extensions of our main target platform, Intel x86 platforms (e.g., MMX, AMD's 3D-Now, Pentium III's Streaming SIMD Extensions.) To that end, we have also experimented with a native code generator.

10 Related work

There are many other examples of embedded DSLs, for music, two- and three-dimensional geometry, animation, hardware design, document manipulation, and many other domains. See [14] for an overview and references. In almost all cases, the implementations were interpretive. Several characteristics of functional programming languages that lend themselves toward the role of host language are enumerated in [7].

Kamin's work on embedded languages for program generation is in the same spirit as our own [17]. As in our approach, Kamin uses host language functions

and tuples to represent the embedded language's functions and tuples, and he uses overloading so that the generators look like the code they are generating. His applications use a functional host language (ML) and generate imperative programs. The main difference is that Kamin did not perform optimization or CSE. Both would be difficult, given his choice of strings to represent programs.

Leijen and Meijer's HaskellDB provides an embedded language for database queries and an implementation that compiles search specifications into optimized SQL query strings for further processing [20]. After exploring several unsuccessful designs, we imitated their use of an untyped algebraic data type and a phantom type wrapper for type-safety.

Thiemann and Sperber made elegant use of Haskell (then Gofer) type classes to make function definitions that are so overloaded that any argument may be either static or dynamic [24]. Their work does not appear to have included simplification of expressions, but could be easily extended to do so. Handles dynamic recursion as well as static, while our approach handles only static recursion. The most notationally awkward aspect of their approach seems to be the use of explicit fixpoint operator.

The Hawk project used overloading to give allow symbolic interpretation of expressions of microprocessor simulation, and performed some simple algebraic simplifications along the way [4]. Like Pan, their expressions were simplified bottom-up.

Our approach to compiling embedded languages can be regarded as an instance of *partial evaluation*, which has a considerable literature (see, e.g., [12, 16]). In this light, our compiler is a handwritten *cogen* (as opposed to one generated automatically through self-application). The main contrasting characteristic of our work is the embedding in a strongly typed meta-language (Haskell). This embedding makes particular use of Haskell type-class-based overloading so that the concrete syntax of meta-programs is almost identical to that of object-programs, and it achieves inlining for free (perhaps too much of it). It also exploits meta-language type inference to perform object-language type inference (except on the optimization rules, which are expressed at the type-unsafe level). Another closely related methodology is multi-stage programming with explicit annotations, as supported by MetaML [22], a polymorphic statically-typed meta-language for ML-style programs.

FFTW is a successful, portable C library for computing discrete Fourier transforms of varying dimensions and sizes [9]. Its numerical procedures are generated by a special purpose compiler, *fftgen*, written in Objective Caml and are better in almost all cases than previously existing libraries. The compiler has some of the same features as our own, performing some algebraic simplification and CSE. One small technical difference is that, while fftgen does memoized simplification, our compiler does bottom-up simplifying construction. It appears that the results are the same. Because the application domain is so specialized, fftgen is more focused than our compiler. In contrast, we wanted to apply our compiler to several domains.

Veldhuizen and others have been using advanced C++ programming techniques to embed a simple functional language into C++ *types* [25, 26]. Functional evaluation is done by the C++ compiler during type-checking and template instantiation. Code fragments specified in inlined static methods are chosen and combined at compile-time to produce specialized, optimized low-level code.

11 Future work

More efficient and powerful rewriting. Our optimizer uses a simple syntactic approach to rewriting. To obtain better results, rewriting and CSE should make use of associative-commutative (AC) matching and comparison, respectively, while still exploiting representation sharing, which is critical for compile-time efficiency.

CSE cleans up after inlining, recapturing what sharing still remains after rewriting. However, where inlining does *higher-order* substitution (in the case of functions), CSE is only first-order, so higher-order redundance remains. Ideally, inlining, if-floating, and CSE would all work cooperatively and efficiently with rewriting. Inlining and if-floating would happen only where rewarded with additional rewrites. Fundamentally, this cooperation seems precluded by the embedded nature of the language implementation, which forces full inlining as the first step, before the DSEL compiler gets to look at the representation.

Invisible compilation. The techniques described in this paper turn compositional specifications into efficient implementations. Image editing applications also allow non-programmers to manipulate images by composing operations. Imagine that such an application were to use abstract syntax trees as its internal editable representation and invisibly invoke an incremental optimizing compiler in response to the user's actions. Then a conventional point-and-click user interface would serve as a "gestural concrete syntax". The display representation would then be one or more bitmaps augmented by custom-generated machine code.

Embeddable compilation. By embedding our language in Haskell, we were able to save some of the work of compiler implementation, namely lexing, parsing, type checking, supporting generic scalar types, functions and tuples. However, it should be possible to eliminate still more of the work. Suppose that the host language's compiler were extended with optimization rules so that it could work much like the one described in this paper. We tried precisely this approach with GHC [23], with partial success. The main obstacle was that the compiler was too conservative about inlining and rewriting. It takes care never to slow down a program, whereas we have found that it is worth taking some backward steps in order to end up with a fast program in the end. Because we do not (yet) work with recursively defined images, laziness in a host language appears not to be vital in this case. It might be worthwhile to try the exercise with an ML compiler.

Plug-and-play code motion. Although we have found the attribute grammar style convenient for merging all code motion transformations in a single pass, the details can become rather tricky. It would be desirable, therefore, to be able to specify new optimizations in the specification style of Section 8, making use of higher-order matching to identify subterms that satisfy certain criteria with respect to binding. The side conditions could be formalised in a variant of temporal logic, as suggested in [21]. We are looking into compilation of such high-level descriptions of transformation rules into attribute grammars. The result would make the code motion phase of the compiler framework described in this paper much more amenable to experimentation.

12 Conclusions

Embedding is an easy way to design and implement DSLs, inheriting many benefits from a suitable host language. Most such implementations tend to be interpretive, and so are too slow for computationally intensive domains like interactive image processing. Building on ideas from Kamin and from Leijen and Meijer, we have shown how to replace embedded interpreters with optimizing compilers, by using a set of syntax-manipulating base types. The result is much better performance with a very small impact on the languages. Moreover, these base types form a reusable DSL compiler framework with which an embedded DSL interpreter can be turned into a compiler with very small changes (thanks to overloading). In our Pan compiler, the rewriting-based optimizations helped speed considerably, as of course does eliminating the considerable overhead imposed by interpretative implementation.

We have produced many examples with our compiler, as may be seen in [6, 8], but more work is needed to make the compiler itself fast and producing even better code. We hope that the compiler's speed can be improved to the point of invisibility so that it can be used by non-programmers in image editors.

13 Acknowledgements

Brian Guenter originally suggested to us the idea of an optimizing compiler for image processing, and has consulted on the project. Erik Meijer helped to sort out the many representation possibilities and suggested the approach that we now use.

References

1. Alstrup, Harel, Lauridsen, and Thorup. Dominators in linear time. *SICOMP: SIAM Journal on Computing*, 28, 1999.
2. A. Appel. *Modern compiler implementation in ML.* Cambridge University Press, 1998.

3. K. Claessen and D. Sands. Observable sharing for functional circuit description. In P.S. Thiagarajan and R. Yap, editors, *Advances in Computing Science ASIAN'99; 5th Asian Computing Science Conference*, volume 1742 of *Lecture Notes in Computer Science*, pages 62–73. Springer-Verlag, 1999. Extended Version Available.
4. Nancy A. Day, Jeffrey R. Lewis, and Byron Cook. Symbolic simulation of microprocessor models using type classes in haskell. In *CHARME'99 poster session (Bad Herranald, Germany)*, September 1999. `http://www.cse.ogi.edu/PacSoft/projects/Hawk/papers/sym_sim.ps`. Companion tech report with details, examples, and Haskell code (OGI Technical Report CSE-99-005), `http://www.cse.ogi.edu/PacSoft/projects/Hawk/papers/tr99-005.ps`.
5. Oege de Moor and Ganesh Sittampalam. Generic program transformation. In *Proceedings of the third International Summer School on Advanced Functional Programming*, Springer Lecture Notes in Computer Science, 1999. `http://-users.comlab.ox.ac.uk/oege.demoor/papers/braga.ps.gz`.
6. Conal Elliott. A Pan image gallery. `http://research.microsoft.com/~conal/-pan/Gallery`.
7. Conal Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3):291–308, May/June 1999. Special Section: Domain-Specific Languages (DSL). `http://-research.microsoft.com/~conal/papers/tse-modeled-animation`.
8. Conal Elliott. Functional images. `http://research.microsoft.com/~conal/-papers/fip`, submitted for publication, March 2000.
9. Matteo Frigo. A fast Fourier transform compiler. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 169–180, 1999. `http://www.acm.org/pubs/articles/proceedings/pldi/-301618/p169-frigo/p169-frigo.pdf`.
10. Dov Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 185–194, Providence, Rhode Island, 6–8 May 1985.
11. Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, May 1984.
12. John Hatcliff, Torben Mogensen, and Peter Thiemann, editors. *Partial Evaluation: Practice and Theory*, volume 1706. Springer-Verlag, 1999.
13. John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Springer Lecture Notes in Computer Science, Copenhagen, Denmark, July 1998.
14. Paul Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.
15. T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer-Verlag, 1987.
16. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. `http://www.dina.kvl.dk/ sestoft/pebook/pebook.html`.

17. Samuel Kamin. Standard ML as a meta-programming language. Technical report, University of Illinois at Urbana-Champaign, September 1996. `http://www-sal.cs.uiuc.edu/~kamin/pubs/ml-meta.ps`.
18. Samuel Kamin and David Hyatt. A special-purpose language for picture-drawing. In USENIX, editor, *Proceedings of the Conference on Domain-Specific Languages, October 15–17, 1997, Santa Barbara, California*, pages 297–310, 1997. `http://www-sal.cs.uiuc.edu/~kamin/fpic/doc/fpic-paper.ps`.
19. Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–164, March 1966. Originally presented at the Proceedings of the ACM Programming Language and Pragmatics Conference, August 8–12, 1965.
20. Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *2nd Conference on Domain-Specific Languages (DSL)*, Austin TX, USA, October 1999. USENIX. `http://www.cs.uu.nl/people/daan/papers/dsec.ps`.
21. Teodor Rus and Eric Van Wyk. Model checking as a tool used by parallelizing compilers. In *Proceedings of the 2nd Formal Methods for Parallel Processing: Theory and Applications. Workshop held at the 11th International Parallel Processing Symposium*, Geneva Switzerland, April 1-5 1997.
22. Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Journal of Theoretical Computer Science*, 2000. To appear. `http://www.cs.chalmers.se/~taha/publications/journal/tcs00.ps`.
23. GHC Team. The Glasgow Haskell compiler. `http://haskell.org/ghc`.
24. Peter Thiemann and Michael Sperber. Program generation with class. In *GI-Arbeitstagung Programmiersprachen*, Aachen, Germany, September 1997.
25. Todd Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. `http://extreme.indiana.edu/~tveldhui/papers/pepm99.ps`. Reprinted in *C++ Gems*, ed. Stanley Lippman.
26. Todd Veldhuizen. C++ templates as partial evaluation. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*. ACM Sigplan, 1999. `http://extreme.indiana.edu/~tveldhui/papers/pepm99.ps`.

A Optimization example

To illustrate the compilation techniques described in this paper, Figure 5 shows snapshots of a sample animation whose specification and supporting definitions are given in Figure 6. Note that *ImageE* is really a type *constructor*, parameterized over the "pixel" type. Visual images have type *ImageE ColorE*, while what one might call "regions" have type *ImageE BoolE*.

As a building block, *checker* is a Boolean image checker that alternates between true and false on a one-pixel checkerboard. The trick is to convert the pixel coordinates from floating point to integer (using the floor function) and test whether the sum is even or odd.

The *checkerBoard* image function takes a square size s and two colors c_1 and c_2. It chooses between the given colors, depending on whether the input point, scaled down by s falls into a true or false square of *checker*.

To finish the example, *swirlBoard* swirls a black and white checker board, using the *swirling* function defined in Section 4.

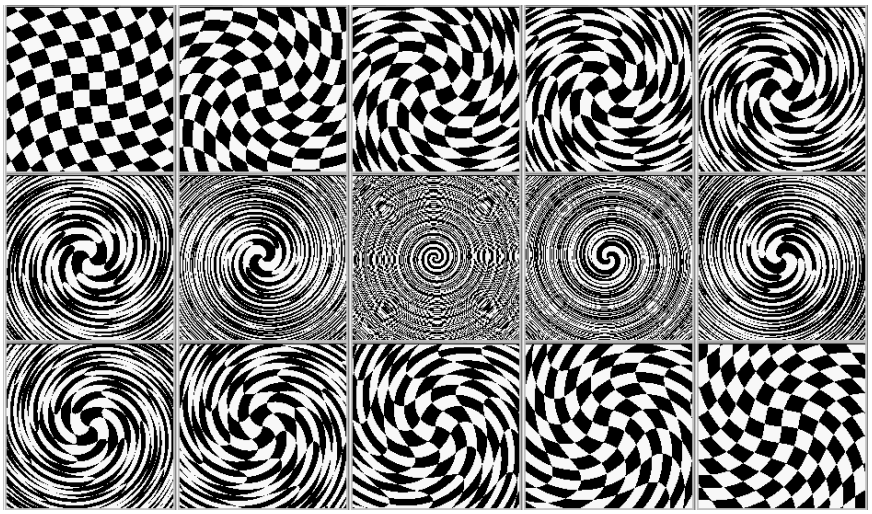

Fig. 5. snapshots of *swirlBoard*, defined in Figure 6

As a relatively simple example of compilation, Figure 7 shows the result of *display swirlBoard* after inlining definitions and performing CSE, but without optimization.

Simplification involves application of a few dozen rewrite rules, together with constant folding, if-floating, and code motion. The result for our example is shown in Figure 8.

Note how the CSE, scalar hoisting, and array promotion have produced three phases of computation. The first block is calculated once per frame of the displayed animation, the second once per line, and the third once per pixel. As an example of the potential benefit of AC-based code motion, note that in the definition of n in Figure 8, the compiler failed to hoist the expression $e * 6.28319$. The reason is simply that the products are left-associated, so this hoisting candidate is not recognized as a sub-expression.

```
swirlBoard :: TimeE → ImageE ColorE
swirlBoard t = swirl (100 * tan t) (checkerBoard 10 black white)

swirl :: Syntactic c ⇒ FloatE → ImageE c → ImageE c
swirl r im = im . swirling r    — Image swirling function

checker :: ImageE BoolE   — Unit square boolean checker board
checker = λ(x, y) → evenE (⌊x⌋ + ⌊y⌋)

checkerBoard :: FloatE → α → α → ImageE α
checkerBoard sqSize c1 c2 =
  ustretch sqSize (cond checker (const c1) (const c2))

— Some useful Pan functions:

cond :: Syntactic a ⇒ BoolE → Exp a → Exp a → Exp a
cond = lift3 ifE   — pointwise conditional
— uniform image stretch
ustretch :: Syntactic c ⇒ FloatE → ImageE c → ImageE c
ustretch s im = im . scale (1/s, 1/s)
```

Fig. 6. Definitions for Figure 5

```
λ t (panX, panY, zoom) (width, height) output →
loop height (λ j →
  loop width (λ i →
    let
      a = 2 π / (100 * sin t / cos t)
      b = −(height `div` 2)
      c = zoom * i2f (j + b) + panY
      d = c * c
      e = −(width `div` 2)
      f = zoom * i2f (i + e) + panX
      g = sqrt (f * f + d) * a
      h = sin g
      k = cos g
      m = 1 / 10
      n = m * (c * k + f * h)
      p = m * (f * k − c * h)
      q = if (⌊p⌋ + ⌊n⌋) .&. 1 == 0 then
            0
          else
            1
      r = ⌊q * 255⌋
      s = 0 <<< 8
      u = output + 4 * j * width
    in
    setInt(u + 4 * i)
          (((s .|. r) <<< 8 .|. r) <<< 8 .|. r)))
```

Fig. 7. Inlined, unoptimized code for Figure 6

```
λ t (panX, panY, zoom) (width, height) output →
let
  a = −(width `div` 2)
  b = mkArr width (λ c → zoom * i2f (c + a) + panX)
  d = −(height `div` 2)
  e = recip (sin t / cos t * 100.0)
in
loop height (λ j →
  let
    f = j * width
    g = zoom * i2f (j + d) + panY
    h = g * g
  in
  loop width (λ i →
    let
      k = (f + i) * 4 + output
      m = readArr b i
      n = sqrt (m * m + h) * e * 6.28319
      p = sin n
      q = cos n
      r = g * q + m * p
      s = m * q + g * −p
    in
    if (⌊s * 0.1⌋ + ⌊r * 0.1⌋) .&. 1 == 0 then
      setInt k 0
    else
      setInt k 16777215))
```

Fig. 8. Optimized version of code from Figure 7