JFP **13** (3): 455–481, May 2003.
DOI: 10.1017/S0956796802004574 Printed in the United Kingdom

Compiling embedded languages

CONAL ELLIOTT
Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA

SIGBJØRN FINNE
Galois Connections, Inc., 3875 SW Hall Blvd., Beaverton, OR 97005, USA

OEGE DE MOOR
Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK

Abstract

Functional languages are particularly well-suited to the interpretive implementations of 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 domain-independent 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 Introduction

The "embedded" approach has proved an excellent technique for specifying and prototyping Domain-Specific Languages (DSLs). Hudak calls such an implementation a "domain-specific embedded language", or DSEL (Hudak, 1998). 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. Much 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, in which domain types are represented as algebraic data types to be "interpreted" by recursive traversal functions, as in Hudak (2000). Worse, these implementations tend to perform redundant computation, because functional languages do not automatically memoize the interpretation functions. This latter problem applies even when domain types are represented as functions rather than

algebraic data types, e.g. the Haskell *geometric region server* (Hudak & Jones, 1994) and Fran (Elliott, 1998).

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*. 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 without the need for the host language's run-time system.

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 DSELs, and handles (a) optimization of expressions over numbers and Booleans, (b) code motion, and (c) code generation. A new DSEL 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 DSEL 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 (Hatcliff *et al.*, 1999).) Generated target language programs are then given to a simple compiler (also implemented in Haskell) for code motion and generation of standalone C code. 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 its target language. (For instance, C's macro language is much weaker than C, and Lisp's macro language *is* Lisp.)

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, such as 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 (Kamin, 1996) with algebraic manipulation.

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

- 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.

While this paper mainly discusses embedded language compilation, a companion paper goes into more detail for the Pan image synthesis language (Elliott, 2001). That paper contains many more visual examples, as does the Pan Gallery (Elliott, 2000).

2 Language embedding

The embedding approach to DSL construction goes back at least to Peter Landin's famous "next 700" paper (Landin, 1966). 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 work required for a new "language" is kept to a minimum. These merits and some drawbacks are discussed, for example. in Elliott (1999) and Hudak (1998).

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 in the form of a simple denotational semantics, as suggested in Kamin & Hyatt (1997, section 3). Consider, for example, the problem domain of image synthesis and manipulation. A simple semantics for images is as functions from continuous 2D space to colors, though it turns out to be very useful to generalize from colors to arbitrary "pixel" types. The representation of colors includes blue, green, red, and opacity ("alpha") components:

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

It is easy to implement operations like image overlay (with partial opacity), assuming a corresponding function, *cOver*, on color values:[1]

$$\begin{aligned}&\mathit{cOver} :: \quad \mathit{Color} \to \quad \mathit{Color} \to \quad \mathit{Color}\\ &\mathit{over} \quad :: \mathit{Image}\ \mathit{Color} \to \mathit{Image}\ \mathit{Color} \to \mathit{Image}\ \mathit{Color}\\ &a\ \text{`}\mathit{over}\text{`}\ b = \lambda p \to a\ p\ \text{`}\mathit{cOver}\text{`}\ b\ p\end{aligned}$$

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

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

[1] Haskell uses backquotes to turn a name, here "*over*" and "*cOver*", into an infix operator. Lambda abstractions are written "$\lambda pat \to exp$" for binding pattern *pat* and body expression *exp*.

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

This model makes it easy to define some familiar warps:

```
type Vector          = (Float, Float)
translate, scale     :: Vector → Warp
rotate               :: Float → Warp

translate (dx, dy) = λ(x, y) → (x + dx, y + dy)
scale (sx, sy)     = λ(x, y) → (sx * x, sy * y)
rotate θ           = λ(x, y) → (x * c − y * s, y * c + x * s)
  where
    c = cos θ
    s = sin θ
```

While these definitions can be directly executed as Haskell programs, we found performance to be inadequate 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 (GHC Team, n.d.). 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 be 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

Figure 1 gives an overview of our system. As indicated in the figure, we implemented some components and used others off the shelf.

In spite of our choice to implement a dedicated compiler, we want to retain most of the benefits of the embedded approach. We resolve this dilemma by applying Kamin's idea of DSELs for program generation (Kamin, 1996) (which is a form of *multi-stage programming* (Taha & Sheard, 2000)). 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 could 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 toextend the base types to support

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

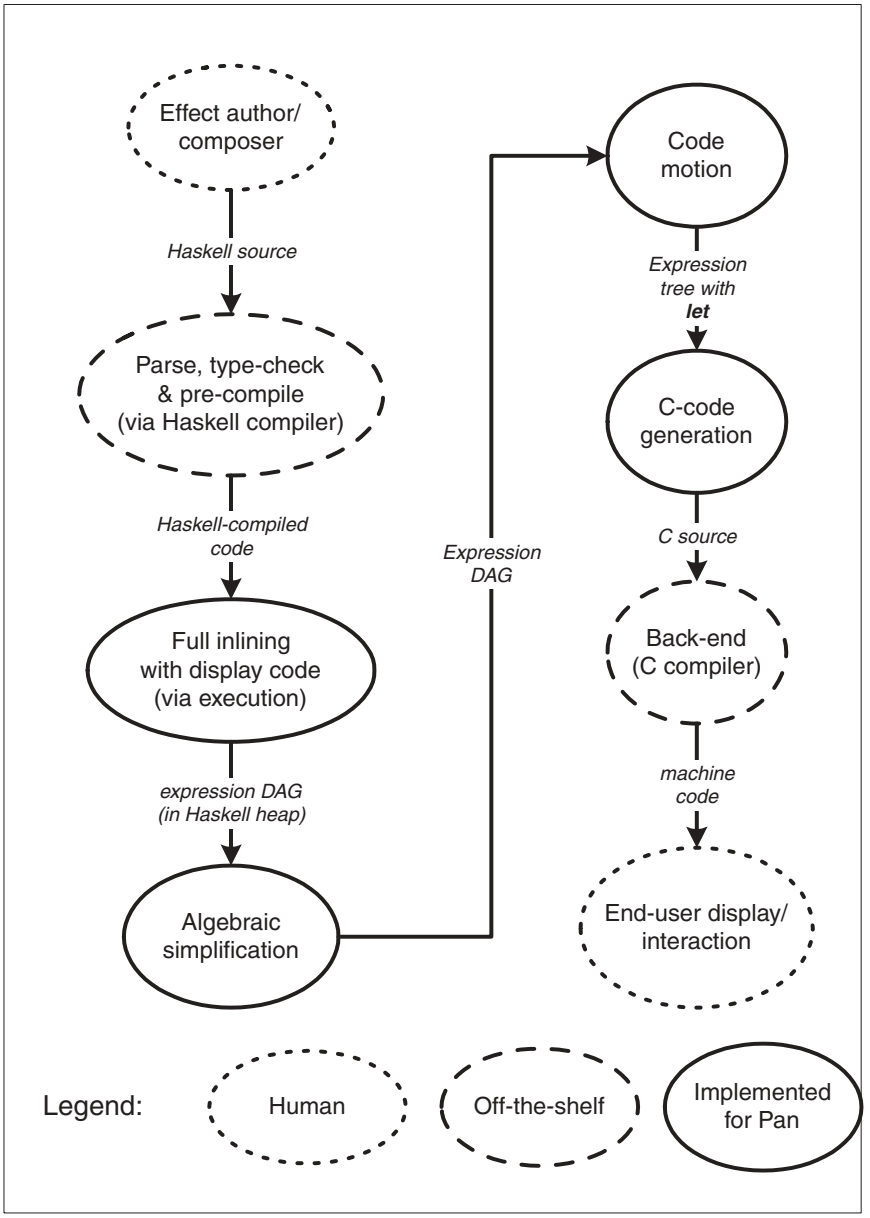

Fig. 1. Overview of compiler structure.

"variables". Then to inspect a function, apply it to a new variable (or tuple of variables as needed), and look at the result. Note that some recursions will cause non-termination in the compiler. This is a drawback of our embedded approach to compilation.

data $FloatE = \ldots | VarFloat\ String$ — named variable

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

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, so that programs can use the most familiar notation for constructing and pattern matching points. Similarly for colors. Thus:

$$\begin{aligned}
&\textbf{type } \mathit{ImageE}\ c &&= \mathit{PointE} \rightarrow c\\
&\textbf{type } \mathit{WarpE} &&= \mathit{PointE} \rightarrow \mathit{PointE}\\
&\textbf{type } \mathit{Point} &&= (\mathit{FloatE}, \mathit{FloatE})\\
&\textbf{type } \mathit{ColorE} &&= (\mathit{FloatE}, \mathit{FloatE}, \mathit{FloatE}, \mathit{FloatE})
\end{aligned}$$

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 ection 2. The *meaning* of these definitions, however, is quite different. We have overloaded the arithmetic operators, trigonometry functions, and a few dozen other functions, to operate on expressions. The *over* function is also defined exactly as before. Assuming that these base types are adequate, an optimizing DSEL compiler is just as easy to implement and extend as with a simple, non-optimizing, embedded, interpretive implementation. Otherwise new syntactic types and/or primitive operators may need to be added, some of which may turn out to be useful for other DSELs.

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 described in section 6, 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 we have not needed any high-level domain rules 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 such as *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 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 redundancy, since each expression type would need its own constructors for variables, binding, and polymorphic and

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

overloaded expression operators (e.g. if-then-else and addition, respectively), as well as its own polymorphic compiler-internal operations on terms.

Instead, we use a single and all-encompassing expression data type *DExp* of "dynamically-typed expressions", rather than the separate types suggested in section 3:

```
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, but rather must edit the single *DExp* type definition. For now we are willing to accept this limitation, but future work may suggest improvements.

The *DExp* representation removes redundancy from representation and supporting code, but it loses type safety. To combine advantages of both approaches, we augment the dynamically-typed representation with the technique of "phantom types" (Leijen & Meijer, 1999). 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, we 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

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

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 are 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, based on the very general type of typ_2. Instead we rely on restricted type signatures, either given explicitly or inherited from the standard Haskell type class declarations.

Other definitions provide a convenient and type-safe primitive vocabulary for *FloatE*. Unfortunately, the *Bool* type is wired into the signatures of operations like $\geqslant$ 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. "<*").

5 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 warp. 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:

```
swirlP :: FloatE → WarpE
swirlP r = λ p → rotate (distO p * (2 π / r)) p

distO :: PointE → FloatE
distO (x, y) = sqrt (x * x + y * y)
```

Inlining *swirlP r* (x, y) yields an expression with much redundancy:

```
(x * cos (sqrt (x * x + y * y) * 2 π / r)
  − y * sin (sqrt (x * x + y * y) * 2 π / r)
,y * cos (sqrt (x * x + y * y) * 2 π / r)
  + x * sin (sqrt (x * x + y * y) * 2 π / r) )
```

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

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* are 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 *swirlP r* were composed with *scale* (u, v) before being applied to (x, y), the two multiplications due to *scale* would each 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 avoids such redundancy, but it incurs considerable overhead of its own, especially if applied at a fine level of granularity, to simple functions. 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), and represents code as graphs, as described in Section 8.

6 Algebraic optimization and smart constructors

An early Pan implementation was based on the Mag program transformation system (de Moor & Sittampalam, 1999). Compilation 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. With this strategy, 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 already-optimized argument term.)

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

- constant-folding;
- if-floating;
- constructor-specific rewrites such as identities and cancellation rules;
- data type constructor application when no optimizations apply; and
- static typing.

As an example, figure 2 shows a smart constructor for conjunction over Boolean expressions. (In fact, the real definition is shorter, because it uses a shared utility function to perform constant folding and if-floating for arbitrary binary smart constructors.) 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.

Pan uses *ifE* for syntactic conditionals, based on an underlying dynamically-typed *ifD*.

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

```
— 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 (orD 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. 2. Simplification rules for conjunction.

```
ifD :: DExp → DExp → DExp → DExp

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
```

The function *ifZ* simplifies redundant or impossible conditions. The statically-typed *ifE* function is overloaded with an overloading instance for expressions:

```
class Syntactic a where ifE :: BoolE → a → a → a

instance Syntactic (Exp a) where ifE = typ3 ifD
```

It would be more convenient to use "if-then-else" syntax, but it is not overloadable in Haskell.

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

```
sin ((if x < 0 then 0 else x) / 2)
```

If-floating without simplification would yield

```
if x < 0 then sin(0/2) else sin(a/2)
```

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

Two constant foldings (0/2 and *sin* 0) result in

if $x < 0$ **then** 0 **else** $sin(a/2)$

Other overloadings for the *Syntactic* class include tuples and functions:

instance (*Syntactic* a, *Syntactic* b) $\Rightarrow$ *Syntactic* (a,b) **where**
  $ifE\ c\ (a,b)\ (a',b') = (ifE\ c\ a\ a',\ ifE\ c\ b\ b')$
— similarly for triples, etc

instance *Syntactic* $b \Rightarrow$ *Syntactic* $(a \rightarrow b)$ **where**
  $ifE\ c\ fa\ fb = \lambda x \rightarrow ifE\ c\ (fa\ x)\ (fb\ x)$

Note that in the case of pairs, the condition c is duplicated (though in the internal Haskell heap representation, there is really only one copy, with two pointers). The code motion phase of compilation (section 8) will replace it with a single evaluation. Similarly, in the case of functions the expression c would get evaluated on each call of the constructed function if not for code motion.

When tuples are consumed to form a single (scalar-valued) expression, if-floating typically causes the redundant conditions to float, forming a cascade of redundant conditionals, which are greatly simplified by *ifZ*. For example, consider the following expression:

let $(x,y) = ifE\ (x < 3)\ (a,b)\ (a',b')$ **in** $x + y$

Using the tuple and *Exp* instance of the *Syntactic* class gives

let $(x,y) = (ifZ\ (x < 3)\ a\ a',\ ifZ\ (x < 3)\ b\ b')$ **in** $x + y$

Substituting for the **let**,

$$ifZ\ (x < 3)\ a\ a' + ifZ\ (x < 3)\ b\ b'$$

Floating the first conditional out of the sum,

$$ifZ\ (x < 3)\ (a + ifZ\ (x < 3)\ b\ b')\ (b + ifZ\ (x < 3)\ b\ b')$$

Floating the other conditional,

$$ifZ\ (x < 3)\ (ifZ\ (x < 3)\ (a+b)\ (a+b'))\ (ifZ\ (x < 3)\ (a'+b)\ (a'+b'))$$

The *ifZ* function then simplifies away both inner conditionals, taking advantage of the outer conditional. The final result is

$$ifZ\ (x < 3)\ (a+b)\ (a'+b')$$

If-floating is another source of code replication, sometimes a great deal of it. Code motion factors out the "first-order" replication, i.e., multiple occurrences of expressions, as with e_2 for the first if-floating clause in figure 2. There is also a *second-order* form of replication, as seen in the *sin* example 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.

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

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 the simplified version to yield a well-defined result when *e* does not. This could happen for example when a boolean expression *e* && * *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 DSEL 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 $PointE \rightarrow ColorE$ 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 prevents 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 times to images. (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 3, 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 *at compile time*. This compile-time application allows the 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 optimized recomputations when only some display parameters change, thanks to a simple dependency analysis.)

The *ActionE* type, as occurs in the definition of *DisplayFun* in Figure 3, represents an action that yields no value, much like Haskell's type *IO* (). It is represented by a

[2] For a high-quality 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.

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

```
type TimeE      = FloatE
type Anim c     = TimeE → ImageE c
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 ColorE → DisplayFun
display anim = λt (panX, panY, zoom) (w, h) output →
  loop h (λj →
    loop w (λi →
      setInt (output + 4 * (j * w + i)) (
        toBGR24 (
          anim t (
            zoom * i2f (i − w `div` 2) + panX,
            zoom * i2f (j − h `div` 2) + panY )))))
```

Fig. 3. Animation display function.

small number of *DExp* constructors and corresponding statically-typed, optimizing wrapper functions, including *setInt* and *loop*. *setInt* takes a memory address (represented as an integer) and an integer value, and writes the value into memory. *loop* 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 loops over j (vertically) and i (horizontally), and sets the appropriate member of its output array to a four-byte (thus multiplication by four) color value. Aside from calculating the destination memory address, the inner loop body samples the animation, *anim*, 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 that eliminate redundant computations, as described in Section 8 and illustrated in Appendix A.

8 Code motion

Once all the above optimizations have been applied, the resulting Haskell data structure turns out to be a directed acyclic graph. This dag is an artifact of Haskell execution, and the sharing is a direct consequence of our bottom-up approach to rewriting (section 6). It represents a rather large expression tree, as explained in section 5. Nodes in the graph with more than one parent represent expressions

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

that were replicated during the inlining and rewriting process. We wish to make the sharing structure explicit using let-bindings and then apply a number of global optimizations (in particular code motion).

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 → (Graph, Node)
dagToLet :: (Graph, Node) → DExp

share :: DExp → DExp
share = dagToLet ∘ expToDag
```

We require that for all e, *share* 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 the target language's "*let*" has strict semantics.

Unfortunately, to implement the *expToDag* transformation in Haskell we had to use non-declarative pointer manipulation. It might be possible to apply the work of Claessen and Sands to avoid this ugly departure from a declarative implementation, or at least make it as innocuous as possible (Claessen & Sands, 1999). 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 = ... | Let Id DExp DExp

type Id = String
```

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.

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 4. 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 5. 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.

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

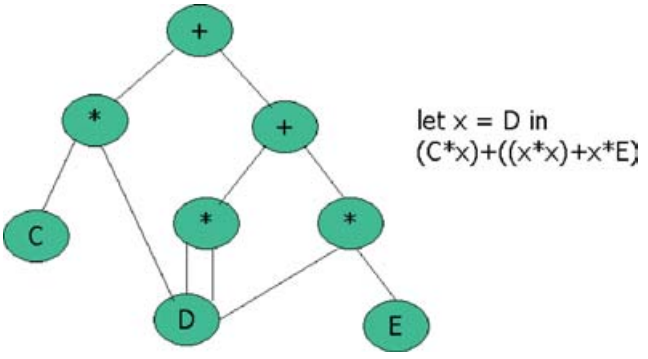

Fig. 4. A dag and the corresponding let-expression.

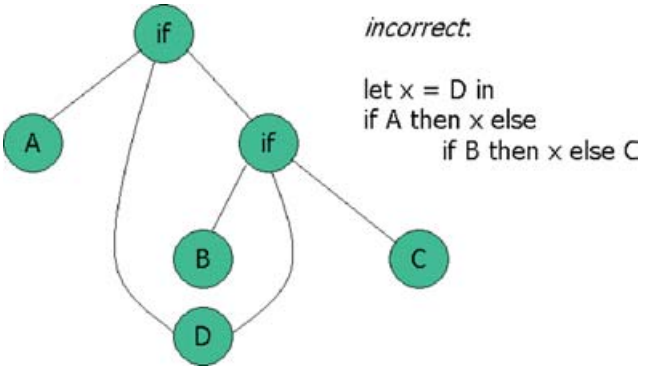

Fig. 5. A dag and an incorrect let-expression.

The above considerations motivate 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

1. At least two children of m contain D.
2. If the expression rooted at m is evaluated, so is at least one of the occurrences of D below it.
3. 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 clause 2 for shared subexpressions D that cannot diverge in the evaluation of m. It only pays to abstract such expressions D if they arevery large,

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

or costly to evaluate. In our compiler, it turned out to be quite a big improvement to hoist `every` non-divergent computation. 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.

Some readers may wonder how we can end up with shared subexpressions that cannot be abstracted to a common *Let*. 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.

There exist algorithms for finding the lowest common ancestors of all repeated subexpressions in time $\mathcal{O}(n \cdot \alpha(n))$ (where there are n nodes in the original graph, and α is the inverse of the Ackermann function) (Harel & Tarjan, 1984; Alstrup *et al.*, 1999), or even $\mathcal{O}(n)$ (Harel, 1985). 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.

See de Moor & Secher (2001) for a more detailed definitions and algorithms for conversion of dags to lets.

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 (Johnsson, 1987). By writing this transformation as an attribute grammar, we can apply it simultaneously

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

with the optimization of array expressions, discussed in the next subsection. This application of attribute grammars is more commonly known to functional programmers as the application of the tupling transformation (Chin, 1993; Pettorossi, 1984), and the introduction of circular data structures (Bird, 1984).

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 similarly.

The type of untyped expressions contains two constructors related to arrays: one to construct a new array, and one 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, which is done through application to a fresh variable (generated via a standard *unsafePerformIO* trick):

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

There is a statically typed version of *ReadArr*:

```
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    — if 0 ⩽ i < size
```

Our first optimization corresponds to the hoisting of loop-invariant code in traditional compilers (e.g. see Appel (1998)). If a subexpression of the body of an array construction does not depend on the index variable, it can be precomputed:[3]

```
  mkArray size (λ i → A[subexp])
=
  let x = subexp in mkArray size (λ i → A[x])
```

subject toseveral side 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".

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

1. The variable i does not occur freely in *subexp*.
2. The *size* of the array must be greater than 0.
3. A is "strict", i.e. every evaluation of the expression matching $A[subexp]$ involves an evaluation of *subexp*.
4. The expression *subexp* is not trivially small.
5. The expression $A[x]$ does not contain *subexp* as a subexpression (so we are abstracting all occurrences of *subexp*).
6. The subexpression *subexp* is not contained in another subexpression of $A[subexp]$ 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 *subexp* 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. This type of tabulation is very common in the optimisation of recursive programs (e.g. see Boiten (1992)).

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

Again there are several side conditions:

1. The variable i does not occur freely in *subexp*.
2. The array sizes $size_1$ and $size_2$ are greater than 0.
3. B is strict.
4. The subexpression *subexp* is not trivially small.
5. The expression $B[y]$ does not contain *subexp* as a subexpression.
6. The subexpression *subexp* is not contained in another sub-expression of B that satisfies the above conditions.

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 3. See Appendix A for an example. (It would be more orthogonal to provide a simpler *mkArray* that only allocates the array, leaving another operation to fill it in, via *loop*. The challenge then would be to use something like Haskell's *runST* (Launchbury & Peyton Jones, 1994) to convert from *Action* back to a purely functional type, perhaps also distinguishing between mutable and immutable arrays, somehow without an expensive copy operation.)

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

9 Code generation

Having performed code motion and loop hoisting, we are ready 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* could be converted into either a C function or native code. Generating C is reasonably straightforward, but requires a little bit of care in places. For instance, we need to account for the fact that C lacks expression-level variable binding support. Also, 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. It cannot, therefore, 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.

Generating C allows us to reuse the strengths of a C code generator, although 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.)

One interesting question is whether a good optimizing C compiler could handle all of the necessary algebraic optimization and code motion. The benefit would be further simplification of our own compiler. We performed an experiment toward answering this question, by turning off our own optimizations and leaving the C compiler optimizations set to maximum. There was a very significant loss of performance, though we do not know precisely why.

10 Related work

There are many other examples of DSELs, for music, two- and three-dimensional geometry, animation, hardware design, document manipulation, and many other domains. See Hudak (1998) 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 Elliott (1999).

Kamin's work on embedded languages for program generation is in the same spirit as our own (Kamin, 1996). 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 (Leijen & Meijer, 1999). 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 make 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 (Thiemann & Sperber, 1997). Their work does not appear to include simplification of expressions, but could be easily extended to do so. It 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 uses overloading to give allow symbolic interpretation of expressions of microprocessor simulation, and performes some simple algebraic simplifications along the way (Day *et al.*, 1999). Like Pan, their expressions are simplified bottom-up.

Our approach to compiling embedded languages can be regarded as an instance of *partial evaluation*, which has a considerable literature (e.g. see Hatcliff *et al.* (1999) and Jones *et al.* (1993)). In this light, our compiler is a handwritten *cogen* (as opposed to one generated automatically through self-application). Berlin, Weise and others have used graphs to represent code (Berlin, 1989; Berlin & Weise, 1990; Weise *et al.*, 1991), thus avoiding unwieldy code duplication, in a somewhat similar way to what we described in Section 8. Danvy also uses smart constructors for optimizing some function applications, in which at least one argument is dynamic (Danvy, 1998). In the context of partial evaluation, 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 (Taha & Sheard, 2000), 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 (Frigo, 1999). 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 (though we have not yet done so).

Veldhuizen and others use advanced C++ programming techniques to embed a simple functional language into C++ *types* (Veldhuizen, 1995, 1999). This language

is executed 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, while inlining does *higher-order* substitution (in the case of functions), CSE is only first-order, so higher-order redundancy 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 (GHC Team, n.d.), 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 our domain. 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

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

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 Rus & Van Wyk (1997). 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.

Haskell hospitality. In some places, Pan is not able to hide the synthetic nature of its types. For instance, the programmer must use "$>*$ and "*ifE*" instead of "$>$" and "*if* $-$ *then* $-$ *else*". The former could probably be fixed by altering the standard Haskell prelude, exploiting "multi-parameter type classes" (Peyton Jones *et al*., 1997). The latter would in addition need a small change to the language itself.

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 DSEL compiler framework with which a simple interpretive DSEL implementation 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 Elliott (2000a, 2000b), but more work is needed to make the compiler itself faster and produce 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.

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, including the use of "phantom types".

A Optimization example

To illustrate the compilation techniques described in this paper, figure A 1 shows snapshots of a sample animation whose specification and supporting definitions are given in figure A 2.

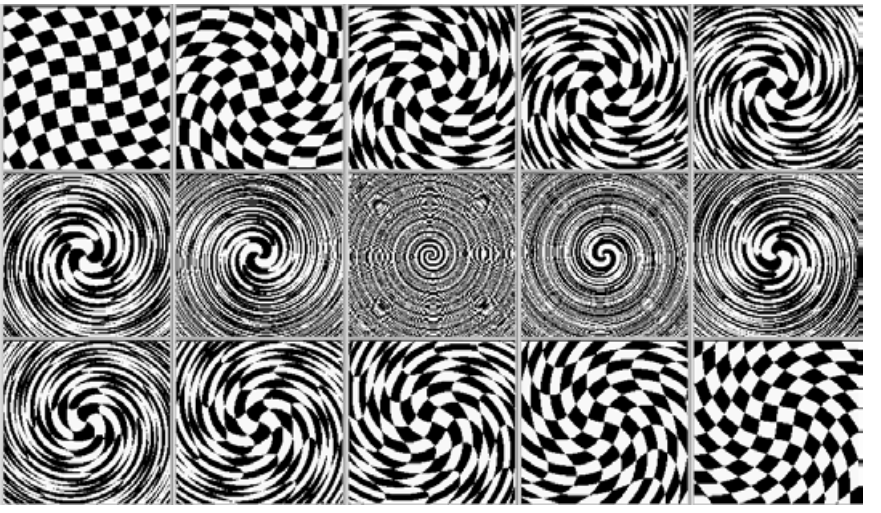

Fig. A 1. snapshots of *swirlBoard*, defined in figure A 2.

```
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 . swirlP r

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

checkerBoard :: FloatE → c → c → ImageE c
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. A 2. Definitions for figure A 1.

As a building block, *checker* is a Boolean image 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.

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

```
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. A 3. Inlined, unoptimized code for figure A 2.

The *checkerBoard* image function takes a square size s and two values c_1 and c_2. It chooses between the given values, 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 *swirlP* function defined in section 5.

As a relatively simple example of compilation, figure A 3 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 A 4.

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 A 4, 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.

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

```
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. A 4. Optimized version of code from figure A 3.

References

Alstrup, S., Harel, D., Lauridsen, P. W. and Thorup, M. (1999) Dominators in linear time. *SICOMP: SIAM J. Comput.* **28**.

Appel, A. (1998) *Modern Compiler Implementation in ML*. Cambridge University Press.

Berlin, A. (1989) *A compilation strategy for numerical programs based on partial evaluation.* Technical Report AITR-1144, Massachusetts Institute of Technology.

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

Bird, R. S. (1984) Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, **21**(3), 239–250.

Boiten, E. A. (1992) Improving recursive functions by inverting the order of evaluation. *Sci. Comput. Program.* **18**(2), 139–179.

Chin, W. (1993) Towards an automated tupling strategy. *Proceedings ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp. 119–132.

Claessen, K. and Sands, D. (1999) Observable sharing for functional circuit description. In: Thiagarajan, P. S. and Yap, R., editors, *Advances in Computing Science ASIAN'99; 5th Asian Computing Science Conference: Lecture Notes in Computer Science 1742*, pp. 62–73. Springer-Verlag.

Danvy, O. (1998) Online type-directed partial evaluation. *Third Fuji International Symposium on Functional and Logic Programming, FLOPS '98 Proceedings*, pp. 271–295. World

https://doi.org/10.1017/S0956796802004574 Published online by Cambridge University Press

Scientific. (Extended version at http://www.brics.dk/RS/97/Ref/BRICS-RS-97-Ref/-BRICS-RS-97-Ref.html#BRICS-RS-97-53.)

Day, N. A., Lewis, J. R. and Cook, B. (1999) Symbolic simulation of microprocessor models using type classes in Haskell. *CHARME'99 poster session*, Bad Herranald, Germany. 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.

de Moor, O. and Secher, J. P. (2001) *Common subexpression elimination of conditional expressions*. Submitted.

de Moor, O. and Sittampalam, G. (1999) Generic program transformation. *Proceedings 3rd International Summer School on Advanced Functional Programming*. http://users.comlab.ox.ac.uk/oege.demoor/papers/braga.ps.gz.

Elliott, C. (1998) Functional implementations of continuous modeled animation. *Proceedings PLILP/ALP*. Springer-Verlag.

Elliott, C. (1999) An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE Trans. Softw. Eng.* **25**(3), 291–308.

Elliott, C. (2000) *A Pan image gallery*. http://research.microsoft.com/~conal/-pan/Gallery.

Elliott, C. (2001) Functional image synthesis. In: Sarhangi, R. and Jablan, S., editors, *Proceedings Bridges 2001, Mathematical Connections in Art, Music, and Science*. http://research.microsoft.com/~conal/papers/bridges2001. An extended version, submitted for publication, is available as http://research.microsoft.com/~conal/-papers/functional-images.

Frigo, M. (1999) A fast Fourier transform compiler. *Proceedings ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pp. 169–180. http://www.acm.org/pubs/articles/proceedings/pldi/301618/p169-frigo/-p169-frigo.pdf.

GHC Team *The Glasgow Haskell compiler*. http://haskell.org/ghc.

Harel, D. (1985) A linear time algorithm for finding dominators in flow graphs and related problems. *Proceedings 17th Annual ACM Symposium on Theory of Computing*, pp. 185–194.

Harel, D. and Tarjan, R. E. (1984) Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.* **13**(2), 338–355.

Hatcliff, J., Mogensen, T. and Thiemann, P. (editors) (1999) *Partial Evaluation: Practice and theory*. Vol. 1706. Springer-Verlag.

Hudak, P. (1998) Modular domain specific languages and tools. In: Devanbu, P. and Poulin, J., editors, *Proceedings: Fifth International Conference on Software Reuse*, pp. 134–142. IEEE Press.

Hudak, P. (2000) *The Haskell School of Expression: Learning functional programming through multimedia*. Cambridge University Press.

Hudak, P. and Jones, M. P. (1994) *Haskell vs. Ada vs. C++ vs. Awk vs. ... an experiment in software prototyping productivity*. Technical report, Yale.

Johnsson, T. (1987) Attribute grammars as a functional programming paradigm. In: Kahn, G., editor, *Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 274*, pp. 154–173. Springer-Verlag.

Jones, N. D., Gomard, C. K. and Sestoft, P. (1993) *Partial evaluation and automatic program generation*. International Series in Computer Science: Prentice Hall International. http://www.dina.kvl.dk/ sestoft/pebook/pebook.html.

Kamin, S. (1996) *Standard ML as a meta-programming language*. Technical report, University of Illinois at Urbana-Champaign. `http://www-sal.cs.uiuc.edu/~kamin/-pubs/ml-meta.ps`.

Kamin, S. and Hyatt, D. (1997) A special-purpose language for picture-drawing. *Proceedings Conference on Domain-Specific Languages*, pp. 297–310. `http://www-sal.cs.uiuc.edu/-~kamin/fpic/doc/fpic-paper.ps`.

Landin, P. J. (1966) The next 700 programming languages. *Comm. ACM*, **9**(3), 157–164.

Launchbury, J. and Peyton Jones, S. L. (1994) Lazy functional state threads. *Proceedings ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 24–35.

Leijen, D. and Meijer, E. (1999) Domain specific embedded compilers. *2nd Conference on Domain-Specific Languages (DSL)*, Austin, TX. `http://www.cs.uu.nl/people/daan/-papers/dsec.ps`.

Pettorossi, A. (1984) *Methodologies for transformations and memoing in applicative languages*. PhD thesis, University of Edinburgh, Scotland.

Peyton Jones, S., Jones, M. and Meijer, E. (1997) Type classes: exploring the design space. *Haskell workshop*.

Rus, T. and Van Wyk, E. (1997) Model checking as a tool used by parallelizing compilers. *Proceedings 2nd Formal Methods for Parallel Processing: Theory and Applications*.

Taha, W. and Sheard, T. (2000) MetaML: Multi-stage programming with explicit annotations. *Theor. Comput. Sci.* **248**(1-2).

Thiemann, P. and Sperber, M. (1997) Program generation with class. *GI-Arbeitstagung Programmiersprachen*.

Veldhuizen, T. (1995) Expression templates. *C++ Report*, **7**(5), 26–31. `http://extreme.-indiana.edu/~tveldhui/papers/pepm99.ps`. Reprinted in *C++ Gems*, ed. Stanley Lippman.

Veldhuizen, T. (1999) C++ templates as partial evaluation. *Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'99)*. `http://extreme.indiana.edu/-~tveldhui/papers/pepm99.ps`.

Weise, D., Conybeare, R., Ruf, E. and Seligman, S. (1991) Automatic online partial evaluation. In: Hughes, R. J. M., editor, *Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 523*, pp. 165–191. Springer-Verlag.