Available online at www.sciencedirect.com
SCIENCE DIRECT
Science of Computer Programming 62 (2006) 47–65
Science of Computer Programming
www.elsevier.com/locate/scico

Combining partial evaluation and staged interpretation in the implementation of domain-specific languages

Christoph A. Herrmann[a,*], Tobias Langhammer[b,1]

[a] *University of Passau, Innstr. 33, Room 128, 94032 Passau, Germany*
[b] *Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB), Department Computer Science Research, Takustr. 7, Room 3154, 14195 Berlin-Dahlem, Germany*

Received 1 January 2005; received in revised form 1 October 2005; accepted 1 February 2006
Available online 12 June 2006

Abstract

We propose a combination of partial evaluation and staged interpretation with MetaOCaml for rapid prototyping of domain-specific languages. Interpretation is an easy way to implement such languages. MetaOCaml can eliminate the overhead of interpretation at run-time, if the interpreter is written in a staged form, i.e., takes the source program separate from the input data in a first stage. Partial evaluation of the source program with values known at compile time can further improve the target code performance. Additional aggressive optimizations are possible due to the absence of general recursion. Algebraic simplifications can even achieve binding-time improvements during the online partial evaluation. Our approach both saves the application programmer completely from binding-time considerations and exploits staged interpretation with MetaOCaml for target code generation.

The example domain presented in this paper is image processing, in which the domain-specific language permits the specification of convolution matrices, summations, case distinctions and non-local pixel accesses. All expressions known at compile time are simplified and all remaining expressions are turned into MetaOCaml code parts, which are combined to form the compiled application program.

The example specifications deal with filtering by convolution and iterations in a series of images for wave effects and temperature distribution.

The experimental results show significant speed-ups if online partial evaluation with algebraic simplifications is used for the elimination of interpretation overhead and optimization of code expressions.

Keywords: Binding-time improvement; Compilation; Domain-specific languages; Meta-programming; Partial evaluation; Staged interpretation

1. Introduction

Domain-specific languages (DSLs) support problem-oriented programming in particular application areas, e.g., BLAS in linear algebra, SQL in relational databases, VHDL in logic simulation and HUME for real-time embedded

* Corresponding author. Tel.: +49 851 509 3074; fax: +49 851 509 3092.
E-mail addresses: herrmann@uni-passau.de (C.A. Herrmann), langhammer@zib.de (T. Langhammer).
URLs: http://www.fmi.uni-passau.de/~herrmann (C.A. Herrmann), http://www.zib.de/CSR/ (T. Langhammer).
[1] The author's work in this article was during his time at the University of Passau, Germany. Tel.: +49 30 84185 410; fax: +49 30 84185 311.

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2006.02.002

systems. In contrast to a general-purpose programming language, a DSL is not necessarily Turing complete but focuses on the typical tasks in the domain and exploits its laws.

We use MetaOCaml, the meta-programming extension of Objective Caml, to write a staged interpreter for a DSL and apply it like a compiler. Here, we can take great benefit from MetaOCaml, concerning the optimizing native code generator and efficient imperative language features in the constructed code. Additionally, abstraction features provide us with the type safety of generated code and higher-order functions for the code construction process itself.

Our main contribution is to demonstrate by example how existing compiler technology can be reused in the rapid prototyping of DSL implementations. In particular, we show how to combine an own partial evaluator with the staged interpreter, yielding a new compiler with significantly improved performance of the target code. We believe that this and similar approaches may be interesting for DSLs with a small user base and also form attractive alternatives to existing DSLs which do not perfectly suit the taste of their users. Additionally, our paper contains some interesting implementation techniques: types and binding times of values can change by algebraic simplification during partial evaluation, loops can be partially unrolled, and a cost calculus is used for the containment of code inflation caused by specialization.

Staged interpretation in MetaOCaml and the integration of partial evaluation will be explained in the next section. In Section 3, we position ourselves in the context of related work. We have chosen a DSL for image processing as an example and present it in Section 4. The partial evaluation techniques are presented in Section 5. We present some selected pieces of staged MetaOCaml code of our implementation in Section 6. The entire implementation is available on our project's web page [1]. Section 7 presents three image processing problems and experimental results obtained with our DSL implementation. Finally, Section 8 summarizes the experiences gained.

2. Principles

2.1. The big picture

An interpreter takes a source program and converts input to output data with regard to the source. Often an interpreter can do a lot of useful work before using the input data. Fig. 1 contains the design of our DSL implementation; without the shaded parts, it depicts the structure of an interpreter separated into phases: the interpreter is first applied to the typed abstract syntax tree of the source program, and then to the input data. Shown in the shaded parts are two conceptually independent methods that we implemented.

(1) *Staged interpretation*. Staging annotations in the residual program generator encapsulate code parts which are then converted by a special MetaOCaml operator (run or .!) into either bytecode or native code, depending on the MetaOCaml installation. The run operator also introduces a distinction between compile time and execution time, in a relative perspective, since the MetaOCaml system imposes a run-time code generation.
(2) *Partial evaluation*. Usually, target code is applied multiple times, e.g., in a loop with many iterations (see Section 7), so we should try to shift as much work as possible from execution time to compile time. Input data which is already known at compile time (static input) can be used in a partial evaluation process to simplify the abstract syntax tree of the source program.

We can use all information available at compile time as static data, but we should only select a subset for which the positive effect on the execution time outweighs the negative effect on the time for partial evaluation. We will discuss that later in our example at the beginning of Section 7.

2.2. Staging in MetaOCaml

OCaml [2] is a language which combines functional, imperative and object-oriented features. On the functional side, OCaml provides us with abstraction features like parametric polymorphism, higher-order functions, algebraic data types and pattern matching with guards. Among the imperative features there are loop nests working on mutable arrays, which makes OCaml attractive for matrix-based numerical computations. OCaml programs can be evaluated interactively or compiled to bytecode or native code.

MetaOCaml [3] extends the OCaml language by features for constructing code expressions and executing them in a future stage computation. The expression syntax is extended by the three staging annotations, *brackets*, *escape* and *run*, and the type system is extended by a type constructor named `code`.

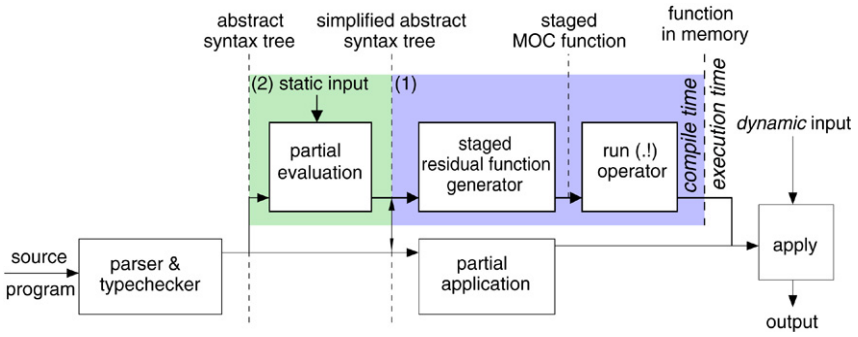

Fig. 1. Components of the DSL implementation.

- *Brackets* (`.< >.`) enclose an object program part.

```
# let fcode = .< fun x -> x*x >.;;
val fcode : ('a, int -> int) code = .<fun x_1 -> (x_1 * x_1)>.
```

The type constructor `code` tells us that `fcode` is not a plain function from `int` to `int` but a piece of object code which acts as such a function when executed. The first component given to the `code` type constructor (`'a`) is a so-called environment classifier. It permits the use of code with unbound variables in the construction process while providing run-time type safety [4].

- The *escape* operator (`.~`) inserts an object program part constructed by the enclosing meta-program into a larger part. Here `fcode`, defined before, is inserted into a code fragment.

```
# let gcode = .< fun y -> (.~fcode) (y+1) >.;;
val gcode : ('a, int -> int) code =
  .<fun y_1 -> ((fun x_1 -> (x_1 * x_1)) (y_1 + 1))>.
```

- The *run* operator (`.!`) compiles a generated object program expression which can be executed immediately.

```
# let gfun = .!gcode;;
val gfun : int -> int = <fun>
# gfun 3;;
- : int = 16
```

Generally, *meta-programming* describes the analysis, transformation and generation of object program parts by meta-program parts. The meta-program parts here are located outside the brackets and are executed at the first stage, before the run operator is applied. At the second stage, the generated code starts execution.

2.3. Partial evaluation

When writing a staged interpreter, the programmer must add staging annotations explicitly, whereas partial evaluation specializes a program automatically, according to program parameters known before the actual execution. This requires a binding-time analysis in order to determine which values are known and which are not. Partial evaluation is classified [5] as either *online* or *offline*, where, in the latter case, the analysis is a separate phase preceding the specialization. The usual mono-variant binding time is a conservative, i.e., pessimistic approximation, if no sophisticated treatment is applied [6]. We opted for the online form, because it can be combined easily with algebraic simplifications incurring binding-time improvements.

Due to the absence of a general form of recursion in our domain-specific setting and simplifications carried out in a bottom-up rewrite process, we need not worry about termination problems. Online partial evaluation simplifies

expressions immediately that do not depend on run-time data. Bottom-up rewriting performs a post-order traversal of the abstract syntax tree of the application program. For partial evaluation and simplification, we prefer a programming language that supports pattern matching with guards, algebraic data types and higher-order functions, although, in principle, any Turing-complete language would do. We want to combine the simplifying partial evaluator directly with the staged interpreter in MetaOCaml to a single DSL compiler which generates type-safe and fast code. Consequently, we took MetaOCaml for the simplifying partial evaluator as well.

3. The context of our work

Interpretation is a well-known technique for embedding a domain-specific language in a general-purpose programming language [7]. The aim of language embedding is to represent objects and operations of the domain by data structures and functions of the host language, to save development time and cost. A preprocessor can be used to transform fancy domain syntax into the syntax of the host language.

Functional languages like Haskell are especially suitable as host languages, because of the availability of higher-order functions. For example, in an embedded DSL for picture generation [8], the abstract syntax tree of a DSL program is used to generate a function in the host language, removing the overhead of interpretation.

The underlying principle is partial evaluation [5] of an interpreter. An interpreter takes a (DSL) program and the current input data. The principle of the so-called first Futamura projection [9] is that partial evaluation of an interpreter delivers a compiled program. This is an important contribution to the theory of program generation. For further historical notes, see Jones [10].

We use partial evaluation first in another context: for simplifying the abstract syntax tree of our domain-specific program. Then, we partially apply the DSL interpreter to the simplified syntax tree. Because the interpreter is staged, we achieve a compiled program, as in the case of the first Futamura projection. The difference to partial evaluation of an interpreter is that, in the case of the staged interpreter, the parts which have to be specialized are exposed syntactically.

There are three attractive functional host languages for DSL embedding, namely MetaOCaml, Template Haskell and C++. The C++ template sublanguage is functional, Turing-complete and can be used to employ a style of programming akin to but not as fancy as Haskell [11]. Czarnecki et al. [12] provide a detailed comparison of these three languages. In our opinion, the advantage of C++ is mainly in the performance of the generated code, whereas the advantages of Template Haskell are the programming comfort of the generator and the generation of code for lazy evaluation.

The big advantage of MetaOCaml is that the generated code is assuredly type-correct if the generating program is type-correct. This is important for us here, since we aim for a run-time code generation, in which type errors in the generated code which have not been caught by the typechecker of the DSL would be unacceptable for the user. Therefore, we avoid Template Haskell, C++ and also those languages that are favored in the partial evaluation community, such as MetaScheme [13].

The guidance provided by the type system of MetaOCaml gives rise to the expectation that the technique of compilation by staged interpretation will become comprehensible by more programmers than before. In a tutorial paper, Taha [14] demonstrated several ways of how to stage an interpreter for arithmetic expressions, regarding error handling and tag elimination, and compared their run-time behavior. We adopted the idea to use environments during the code construction phase, but avoid them in the generated code itself. Also, we try to stay away from the continuation-passing style as long as it is feasible, to make meta-programs easier to read and maintain. Details on the elimination of interpretation overhead are discussed by Taha et al. [15].

Our own previous work with MetaOCaml is on the partial evaluation of recursive skeletons with functional arguments for parallel processing [16]. The background of this work, i.e., domain-specific programming in the field of parallelism, is summarized by Lengauer [17]. The idea of making abstract information explicit in the representation, in favor of code-optimizing transformations, can be found in our work on the size inference of nested lists [18]. In that paper, we applied a change of basis to the list representation, such that length information and indexing are separated from the list content. Attaching abstract information to code expressions is not unusual in a compiler, but it becomes especially elegant when it is combined with transformation rules based on pattern matching, abstract interpretation and partial evaluation [19].

The task graph library [20] is a run-time specialization system based on C++, which can optimize nested `for`-loops according to run-time parameters. An image filtering program is specialized with the entries of a convolution matrix. Another example is a tiling transformation applied to a nest of loops for matrix multiplication, which can optimize the cache behavior significantly. Programs in our DSL are also specialized with the entries of convolution matrices. We took the data of these matrices from the Intel image processing library [21]. Two other tools which deal with convolutions are the GNU image manipulation program GIMP [22] and the image processing library of MATLAB [23]. Another image processing application amenable to partial evaluation is ray tracing, where intersection and shading functions can be specialized with respect to objects and light sources in the scene. Andersen took C-Mix, a partial evaluator for the language C, for this purpose [24]. Other work on specializing shaders in image processing was by Guenther et al. [25].

At least three related approaches are conceptually very close to our work, since they combine the concept of interpretation and partial evaluation. Glück and Jørgensen [26] inserted an interpreter in front of a partial evaluator to change its behavior, where our approach is to insert a partial evaluator into a staged interpreter to change the generated code. Sperber and Thiemann implemented a composition of partial evaluation and compilation in Scheme [27]. Thibault et al. [28] implemented DSLs by the partial evaluation of interpreters to achieve compilation or domain-specific optimization.

One distinguishes between online and offline partial evaluation [5]. The offline form performs a separate binding-time analysis before the specialization and avoids some termination problems in the case of a language with general recursion. We present binding-time rules that could be used for a separate binding-time analysis of our DSL, but implement an online form which does the analysis on the fly with the specialization, making use of binding-time improvements gained by simplification. For the partial evaluation of languages with general recursion, it is interesting to know that offline partial evaluation can be as accurate as online partial evaluation [6].

4. A domain-specific language for image processing

Image processing transforms an image into a modified image, usually generating the output pixel by pixel. An image is given by a mapping of `row`/`col` coordinates to an intensity, for each color channel. A gray-scale image has one channel; a color image usually has three: one for each of the colors, red, green and blue.

The user specifies the intensity of each pixel of the output image by providing either a separate expression for each color channel or a united expression for all channels. The specification of the intensity of each channel can use the intensities of all channels of the original image. If a series of output images is to be generated, the special variable `iter` can be used to refer to a single frame.

4.1. Syntax

We adopt the convention that non-terminals are written in *italics* and terminal symbols appear in `teletype` font.

A program (*toplevelExpr*) consists of a series of local definitions that are common to all output channels and a specification of the intensities of the output channels (*channelSpecs*) for the pixel located at the position given by `row` and `col`. The channel intensities can be addressed either explicitly by a list with one element for each output channel (first alternative of *channelSpecs*) or implicitly by a single expression for all channels (second alternative). The implicit alternative is purely syntactic sugar to support the frequent case that the specifications of all channels are similar; differences can be addressed by referring to the output channel identifier `current`. Desugaring is done by cloning the expression and by prepending each copy with a `let`-binding of `current` to the current list position, e.g.:

```
[ 3 channels: image(row, col, (current + 1) % 3) ]
```

is desugared to

```
[ let current = 0 in image(row, col, (current + 1) % 3);
  let current = 1 in image(row, col, (current + 1) % 3);
  let current = 2 in image(row, col, (current + 1) % 3) ]
```

Aside from `current`, an expression (*expr*) can be a constant (*const*), a variable (*var*), a read access to a channel intensity value of a pixel (`image...`), a parenthesized expression, a simple operator application (*prefix1*, *prefix2*, *infix*),

a conditional expression (`if..then..else`), a local definition (`let..in`), a summation (`sum..of`), a matrix definition (*matrix*) or the selection of a matrix element by indexing (*index*).

The grammar of the language is listed below, with start symbol *toplevelExpr*. Operators have the usual precedences.

toplevelExpr ←
  `let` *var* `=` *expr* `in` *toplevelExpr*
  | *channelSpecs*
channelSpecs ←
  `[` *expr* (`;` *expr*)* `]`
  | `[` *constInt* `channels` `:` *expr* `]`
row ← *expr*
col ← *expr*
channelID ← *expr*
const ← *constBool* | ...
var ← *letter* (*letter* | *digit*)*
prefix1 ← $-^{1}$ | `not` | `floor` | `sin` | ...
prefix2 ← `min` | `max`
infix ← `+` | $-^{2}$ | `*` | `<` | `&&` | ...
constBool ← `false` | `true`

expr ←
  const
  | *var*
  | `current`
  | `image` `(` *row*, *col*, *channelID* `)`
  | `(` *expr* `)`
  | *prefix1 expr*
  | *prefix2* `(`*expr*, *expr*`)`
  | *expr infix expr*
  | `if` *expr* `then` *expr* `else` *expr*
  | `let` *var* `=` *expr* `in` *expr*
  | `sum` *var* `from` *expr* `to` *expr* `of` *expr*
  | *matrix*
  | *index*
matrix ← `[` *constFloat*$^{+}$ (| *constFloat*$^{+}$)* `]`
index ← *var* `[`*expr*, *expr*`]`

4.2. Typing rules

Our typing rules are quite conventional, except for the rules for summation and image and matrix indexing. For brevity, we define rules for entire sets of operators. Extending the type environment Γ with an association of x to τ is written $\Gamma[x := \tau]$. We write $e_1, e_2 : \tau$ for $(e_1 : \tau) \wedge (e_2 : \tau)$. Our language supports four first-order data types: booleans, restricted integers, floating-point numbers, and matrices of floating-point constants.

Constants: Bool constants are `false` and `true`. Matrix constants consist only of `Float` constants and have type `Matrix`. Other constants are either of type `Int` or `Float`. `Float` constants can be distinguished from `Int` constants by their decimal point.

Variables:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Image indexing:

$$\frac{\Gamma \vdash \mathit{row}, \mathit{col}, \mathit{chan} : \texttt{Int}}{\Gamma \vdash \texttt{image}(\mathit{row}, \mathit{col}, \mathit{chan}) : \texttt{Float}}$$

Conditional expression:

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2, e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ : \tau}$$

Local definition:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x := \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ : \tau_2}$$

Summation:

$$\frac{\begin{array}{l}\Gamma \vdash e_1, e_2 : \texttt{Int} \\ \Gamma[x := \texttt{Int}] \vdash e_3 : \tau \\ \tau \in \{\texttt{Int}, \texttt{Float}\}\end{array}}{\Gamma \vdash \texttt{sum}\ x\ \texttt{from}\ e_1\ \texttt{to}\ e_2\ \texttt{of}\ e_3 : \tau}$$

Indexing a matrix:

$$\frac{\Gamma \vdash a : \texttt{Matrix} \quad \Gamma \vdash e_1, e_2 : \texttt{Int}}{\Gamma \vdash a\,[e_1, e_2] : \texttt{Float}}$$

Coercion:

$$\frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash e : \texttt{Float}}$$

Conversion:

$$\frac{\Gamma \vdash e : \texttt{Float}}{\Gamma \vdash \texttt{floor}\,e : \texttt{Int}}$$

Unary operators (except `floor`):

$$\frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma \vdash \texttt{not}\,e : \texttt{Bool}}$$

$$\frac{\Gamma \vdash e : \texttt{Int} \quad \varphi \in \{-^{1}, \texttt{abs}\}}{\Gamma \vdash \varphi\,e : \texttt{Int}}$$

$$\frac{\Gamma \vdash e : \texttt{Float} \quad \varphi \in \{-^{1}, \texttt{abs}, \texttt{sin}, \ldots\}}{\Gamma \vdash \varphi\,e : \texttt{Float}}$$

Binary prefix operators:

$$\frac{\begin{array}{l}\Gamma \vdash e_1, e_2 : \tau \quad \tau \in \{\texttt{Int}, \texttt{Float}\} \\ \varphi \in \{\texttt{min}, \texttt{max}\}\end{array}}{\Gamma \vdash \varphi(e_1, e_2) : \tau}$$

Binary infix operators:

$$\frac{\Gamma \vdash e_1, e_2 : \texttt{Bool} \quad \circledast \in \{\texttt{=}, \texttt{<>}, \texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1 \circledast e_2 : \texttt{Bool}}$$

$$\frac{\Gamma \vdash e_1, e_2 : \texttt{Float} \quad \circledast \in \{\texttt{+}, \texttt{-}^2, \texttt{*}, \texttt{/}, \texttt{**}\}}{\Gamma \vdash e_1 \circledast e_2 : \texttt{Float}}$$

$$\frac{\Gamma \vdash e_1, e_2 : \texttt{Int} \quad \circledast \in \{\texttt{+}, \texttt{-}^2, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{**}\}}{\Gamma \vdash e_1 \circledast e_2 : \texttt{Int}}$$

Infix comparison operators:

$$\frac{\Gamma \vdash e_1, e_2 : \tau \quad \tau \in \{\texttt{Int}, \texttt{Float}\} \quad \circledast \in \{\texttt{=}, \texttt{<>}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1 \circledast e_2 : \texttt{Bool}}$$

4.3. Semantics

As a purely syntactic issue, implicit channel specifications have been described in Section 4.1. The denotational semantics of local definitions in *toplevelExpr*, used to bind expressions which are common to more than one color channel, is the same as if they were moved into each channel expression. Therefore, this subsection deals with expressions (*expr*) only.

The semantics is given informally, where suitable, or by the valuation function

$$\mathcal{I}\ [\![\ e\ ]\!]\ \varepsilon,$$

in which e is an expression and ε is an environment, i.e., a function that maps program variables to program values associated with their type.

Since our language is specialized for the domain of image processing, we dispense with the feature of general recursion in favor of optimization potential. Therefore, the valuation function is stated compositionally and in explicit form, without the need for a treatment of fixed points. The restricted form of recursion induced by summation can formally be defined inductively, but we express it here briefly using the $\sum$ notation for summation.

Initial Environment: The initial environment consists of the current output pixel coordinates (`row`, `col`), image dimensions (`width`, `height`), the iteration counter (`iter`), the color channel numbers (`red` $\mapsto$ 0, `green` $\mapsto$ 1, `blue` $\mapsto$ 2, `gray` $\mapsto$ 0), and the maximum color value per channel (used internally only).

Variables, Conditionals, Local Definitions: Their semantics is conventional.

- Variables:
  $$\mathcal{I}\ [\![\ x\ ]\!]\ \varepsilon = \varepsilon(x)$$
- Conditionals:
  $$\mathcal{I}\ [\![\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ ]\!]\ \varepsilon = \begin{cases} \mathcal{I}\ [\![\ e_2\ ]\!]\ \varepsilon, & \text{if } \mathcal{I}\ [\![\ e_1\ ]\!]\ \varepsilon = \mathsf{True} \\ \mathcal{I}\ [\![\ e_3\ ]\!]\ \varepsilon, & \text{otherwise} \end{cases}$$
- Local Definitions:
  $$\mathcal{I}\ [\![\ \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ ]\!]\ \varepsilon = \mathcal{I}\ [\![\ e_2\ ]\!]\ (\varepsilon[x{:=}\mathcal{I}\ [\![\ e_1\ ]\!]\ \varepsilon])$$

Image indexing: The function `image`(*row*, *col*, *chan*) delivers the input intensity of the pixel at position $(r, c) = (\mathcal{I}\ [\![\ row\ ]\!]\ \varepsilon, \mathcal{I}\ [\![\ col\ ]\!]\ \varepsilon)$ and color channel $\mathcal{I}\ [\![\ chan\ ]\!]\ \varepsilon$. The pixel position must be within the range of the image index space, i.e., $(r, c) \in [0\ ..\ \varepsilon(\texttt{height})-1] \times [0\ ..\ \varepsilon(\texttt{width})-1]$. The result is a float in the interval $[0, 1]$.

Arithmetic and bool operators: The semantics is the same as that of OCaml for ints and bools. For float operators, we use overloading of the int symbols. For example, operator `/` performs an integer division if applied to two ints and a floating-point division if applied to two floats. `%` is the modulo operation on ints and `**` is the power. Int values are coerced when used in place of a float argument. Floats must be converted explicitly, using the `floor` function.

Summation: The aim of a summation is to iterate through a number of points at which expressions are evaluated and add the individual values.

$$\mathcal{I}\ [\![\ \texttt{sum}\ x\ \texttt{from}\ e_1\ \texttt{to}\ e_2\ \texttt{of}\ e_3\ ]\!]\ \varepsilon = \sum_{i=\mathcal{I}\ [\![\ e_1\ ]\!]\ \varepsilon}^{\mathcal{I}\ [\![\ e_2\ ]\!]\ \varepsilon} \mathcal{I}\ [\![\ e_3\ ]\!]\ (\varepsilon[x{:=}i])$$

First, the expressions e_1 and e_2 are evaluated in the existing environment, giving the lower and upper bounds. Then, for each integral point in the range of these bounds, the expression e_3 is evaluated, where x is available as a new local definition carrying the value of the current point.

5. Partial evaluation methods

5.1. Binding times of expressions

Our simplifications need to distinguish between expressions provably static which can be evaluated at compile time and expressions which are potentially dynamic and cannot.

We use a two-value abstract domain for binding times, as described by Jones et al. [5]: S represents static and D dynamic information. We define the order $\mathsf{S} \sqsubset \mathsf{D}$ and the least upper bound $\sqcup$, where:

$$a \sqcup b = \begin{cases} \mathsf{S}, & \text{if } a = \mathsf{S} \wedge b = \mathsf{S} \\ \mathsf{D}, & \text{otherwise.} \end{cases}$$

The binding-time rules of this section define whether expressions are static or dynamic, provided that no binding-time improvements are made afterwards. This leads to binding-time rules which are not too complicated and need not be changed when new optimization rules are added.

During partial evaluation, the binding time of an expression is static iff the recursive simplification has attached some constant static value to it. This information includes all binding-time improvements that have been made during the simplification.

An example binding-time improvement occurs if e_1 is a dynamic value and e_2 is static 0. Then, in the expression

$$\texttt{let } x = e_1 * e_2 \texttt{ in } e_3,$$

the value x will be dynamic although, after optimization to

$$\texttt{let } x = 0 \texttt{ in } e_3,$$

the binding time rules determine x to be static.

Binding-time improvements can be quite complicated: in a summation with static bounds, some of the summands can be static, e.g., annihilated, and the others dynamic. Respecting that the addition of floats is not associative, some of the static summands can still be combined at compile-time.

$\mathcal{B}\ [\![\ e\ ]\!]\ \sigma\ \beta$ denotes the binding time of a program expression e in an environment σ that associates static program variables with their respective values and an environment β that maps program variables to binding times. For conditional expressions, local definitions and summations, we evaluate static values using the valuation function $\mathcal{I}\ [\![\ .\ ]\!]$. discussed above.

Constants: All constants are static values.

Variables: Generally, variables are inserted into the environment by local definitions (`let`) and summations.

The binding time of a locally defined variable is determined by that of the right-hand side of its definition. A summation variable is static if and only if both bounds are static.

Variables x initially *static* ($\beta x = \mathsf{S}$, and x is in the domain of σ) are:

- `width`: the horizontal extension of the input image in pixels
- `height`: the vertical extension of the input image in pixels
- `iter`: the iteration counter indicating the current index of an image in a series to be processed (see also the example in Section 7.3)
- the constants `red`, `green`, `blue` and `gray` to name color channels
- `maxval`: the maximum color value

Variables x initially *dynamic* ($\beta x = \mathsf{D}$) are:

- `row`: the vertical position of the output pixel
- `col`: the horizontal position of the output pixel

Note that the index `current` of the current output channel is already replaced by a `let`-binding in the syntactic desugaring process.

Image indexing: $\mathcal{B}\,[\![\ \texttt{image}(\_,\_,\_)\ ]\!]\,\sigma\,\beta = \mathsf{D}$. The pixel intensity value is not available at the first stage; therefore the result is always dynamic.

Operators:

unary	$\mathcal{B}\,[\![\ \varphi e\ ]\!]\,\sigma\,\beta$	$= \mathcal{B}\,[\![\ e\ ]\!]\,\sigma\,\beta$
binary prefix	$\mathcal{B}\,[\![\ \varphi(e_1, e_2)\ ]\!]\,\sigma\,\beta$	$= \mathcal{B}\,[\![\ e_1\ ]\!]\,\sigma\,\beta \sqcup \mathcal{B}\,[\![\ e_2\ ]\!]\,\sigma\,\beta$
binary infix	$\mathcal{B}\,[\![\ e_1 \circledast e_2\ ]\!]\,\sigma\,\beta$	$= \mathcal{B}\,[\![\ e_1\ ]\!]\,\sigma\,\beta \sqcup \mathcal{B}\,[\![\ e_2\ ]\!]\,\sigma\,\beta$

Conditional expression: If the condition is static, then the binding time of the appropriate branch is taken, otherwise the result value is dynamic. Due to the absence of side-effects, we can remove the condition check if it is static.

$$\mathcal{B}\,[\![\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ ]\!]\,\sigma\,\beta = \begin{cases} \mathcal{B}\,[\![\ e_2\ ]\!]\,\sigma\,\beta, & \text{if } \mathcal{B}\,[\![\ e_1\ ]\!]\,\sigma\,\beta = \mathsf{S} \wedge \mathcal{I}\,[\![\ e_1\ ]\!]\,\sigma = \mathsf{True} \\ \mathcal{B}\,[\![\ e_3\ ]\!]\,\sigma\,\beta, & \text{if } \mathcal{B}\,[\![\ e_1\ ]\!]\,\sigma\,\beta = \mathsf{S} \wedge \mathcal{I}\,[\![\ e_1\ ]\!]\,\sigma = \mathsf{False} \\ \mathsf{D}, & \text{otherwise} \end{cases}$$

Local definition: The binding time of the whole expression is that of the result expression. Therefore, the environment β is extended with the binding time of the local binding. Note that variables tagged as dynamic will not be looked up in σ.

$$\mathcal{B}\,[\![\ \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ ]\!]\,\sigma\,\beta = \begin{cases} \mathcal{B}\,[\![\ e_2\ ]\!]\,\sigma\,(\beta[x{:=}\mathsf{D}]), & \text{if } \mathcal{B}\,[\![\ e_1\ ]\!]\,\sigma\beta = \mathsf{D} \\ \mathcal{B}\,[\![\ e_2\ ]\!]\,(\sigma[x{:=}\mathcal{I}\,[\![\ e_1\ ]\!]\,\sigma])\,(\beta[x{:=}\mathsf{S}]), & \text{otherwise} \end{cases}$$

Summation expression: If at least one of the bounds is dynamic, then the entire expression is dynamic, otherwise the least upper bound of the binding times of all summands in the static range is returned. For example, if summation is used for a convolution, exactly those summands are static for which the entry in the matrix is 0; for the others, the weight is multiplied with some dynamic pixel value.

$$\mathcal{B}\,[\![\ \texttt{sum}\ x\ \texttt{from}\ e_1\ \texttt{to}\ e_2\ \texttt{of}\ e_3\ ]\!]\,\sigma\,\beta$$
$$= \begin{cases} \mathsf{D}, & \text{if } \mathcal{B}\,[\![\ e_1\ ]\!]\,\sigma\,\beta \sqcup \mathcal{B}\,[\![\ e_2\ ]\!]\,\sigma\,\beta = \mathsf{D} \\ \bigsqcup_{i=\mathcal{I}\,[\![\ e_1\ ]\!]\,\sigma}^{\mathcal{I}\,[\![\ e_2\ ]\!]\,\sigma} \mathcal{B}\,[\![\ e_3\ ]\!]\,(\sigma[x{:=}i])\,(\beta[x{:=}\mathsf{S}]), & \text{otherwise} \end{cases}$$

Indexing: Since the matrix entries can only be constant float values, the binding time of the indexing expression is static iff both indices are static.

$$\mathcal{B}\,[\![\ a[e_1,e_2]\ ]\!]\,\sigma\,\beta = \mathcal{B}\,[\![\ e_1\ ]\!]\,\sigma\,\beta \sqcup \mathcal{B}\,[\![\ e_2\ ]\!]\,\sigma\,\beta$$

5.2. Binding times in static evaluation

As the reader may have noticed in the previous subsection, our binding-time functions evaluate static conditions, static local definitions and static loop bounds at specialization time. For simplicity, the definitions there describe the pure binding-time rule portions. In the implementation, the calculation of binding times is intertwined with a static evaluator in a way that static expressions are reduced immediately.

Program optimizations like `0*x`→`0` achieve binding-time improvements (dynamic `0*x` to static `0`) which, for brevity, are not described in the binding-time rules. The following program fragment displays the simplification of multiplication in the case that at least one factor is zero or one. The variables `a` and `b` are the simplified operands and the application `unC x` returns `Some c` iff `x` is a syntax tree node representing the constant `c`. If no simplification can be made, the last pattern recombines `a` and `b` with a multiplication.

```
| MulI | MulF ->
      begin match unC a, unC b with
      | Some av, Some bv           -> exp_of_value (binopfun op av bv)
      | Some av, _ when is_zero av -> zero_of dtype
      | _, Some bv when is_zero bv -> zero_of dtype
      | Some av, _ when is_one av  -> b
      | _, Some bv when is_one bv  -> a
      | _ -> combine a b
      end
```

Function `is_zero` and `is_one` check if a value is zero or one, respectively. Function `exp_of_value` constructs a node of the syntax tree for a constant and `zero_of dtype` a node that represents the zero element of type `dtype`.

5.3. Optimization example: Loop unrolling

An important part of our image processing language is dedicated to filtering, which basically means to sum the weighted intensity values of a fixed number of pixels, usually expressed by a sum operator, i.e., a kind of loop. Since the bodies of such loops are usually small, unrolling the loops could increase performance significantly. To prevent the risk of code explosion and memory overflow incurred by a complete unrolling of large loops, we aim for a partial loop unrolling which limits the generation of chains of plus operations. Partial loop unrolling to optimize pipelining behavior is described in detail by Cohen et al. [29].

The summation operator is the most sophisticated part of our combined analysis/simplification. We distinguish between the following cases:

(1) Both loop bounds are static:
Starting from the lower bound, we evaluate the body for each iteration as long as the body is static and accumulate the results, respecting the non-associativity of floating-point addition. For example, as long as entries in a convolution are weighted with 0, the sum remains a static 0, and the first non-zero weight changes the intermediate sum to a dynamic value. We use this statically computed prefix for the initial value of the sum in the residual program and generate code only for the remaining iterations: the body is simplified independently of an actual iteration, and the remaining iterations are partially unrolled.
(2) At least one loop bound is dynamic:
The body expression of the summation is simplified without knowledge of the value of the index variable.[2] At this point, we are unrolling the loop partially too, expecting to gain performance by reducing the number of boundary checks.

For unrolling an expression like

sum i **from** lo **to** up **of** $body(i)$

we use the law

$$I = \lfloor I/N \rfloor \cdot N + (I \bmod N)$$

with $(up - lo + 1)$ for the number I of iterations to obtain the following semantically equivalent loop template:

sum k **from** 0 **to** $((up - lo + 1)$ **div** $N) - 1$ **of**
  let $b = lo + k * N$ **in**
  sum s **from** 0 **to** $N - 1$ **of**
    $body(b + s)$
$+$
sum t **from** $((up - lo + 1)$ **div** $N) * N + lo$ **to** up **of**
  $body(t)$

The inner loop on s computes the sum of blocks of size N and is going to be expanded to a chain of additions. The choice of N depends on the limit of the expanded code that we are willing to accept. The amount of syntax tree leaves of each expanded sum is limited to a global constant M. In our application, we set the constant to 2000. Let E be the number of syntax tree leaves of *body*. Then we choose $N = \lfloor M/E \rfloor - 1$ and do the unfolding iff $N > 1$.

The sum of s of this template can and will always be unrolled, no matter what the binding times of the summation bounds are. In the case that both lo and up are static, the sum of t will also be unrolled because its number of operations is limited to N. Furthermore, if the original sum has static bounds with a range of less than N summations, it will be completely unrolled. The reason for this is that the sum of k will have an empty range and thus will be eliminated. So, only the unrolled sum of t remains.

[2] An error in the body makes the simplification fail, even if the number of operands of the sum is zero at run-time. Since static expressions in the loop body cannot depend on a dynamic value, all the body evaluation can ever do is fail. This warrants an outright rejection of the program.

6. Implementational aspects

We focus on the central issues of our implementation here. For a complete overview, the reader may refer to [30]. The program sources can be obtained from our meta-programming project's web page [1].

6.1. Data types

Our language supports four data types: booleans, restricted integers, floating-point numbers, and matrices of floating-point constants, which are to be used as lookup-tables for convolutions.

```
type dtype = Bool | Int | Float | Matrix
type value = VBool of bool | VInt of int |...
```

An operator (`op`) can be a constant (`C`), a variable (`V`), a query operation for a pixel intensity (`Image`), a conversion from `Int` to `Float` (`Int2Float`) or back (`Floor`), a simple unary or binary operation (`UnOp`, `BinOp`), a conditional expression (`If`), a local definition (`Let`), an access to an entry of a constant matrix (`IndexMatrix`) or a summation (`Sum`).

```
type op = C of value      | V of string | Image
        | Int2Float       | Floor       | UnOp  of unOp
        | BinOp of binOp  | If          | Let of string
        | Sum of string   | IndexMatrix of string
```

The string argument of `Let` and `Sum` carries the name of the local variable, which can be accessed by `V` if it is a scalar value, or by `IndexMatrix` in the case of a matrix. Since, aside from the current operator, the nodes of the expression tree also contain the subtrees and the type information, the constructors that represent the operators need only to be parameterized with the remaining information, e.g., the name of a variable.

The following constructors of types `unOp` and `binOp` describe simple unary and binary operators. In order to avoid an additional choice over types in the pattern matching, there are different constructors for overloaded operators. In case of overloading, the constructor for the int operator ends with `I`, while that for the float operator ends with `F` and that for the bool operator ends with `B`.

```
type unOp  = NegI | NegF | Not  | AbsI | AbsF ...
type binOp = AddI | SubI | MaxI | AddF | SubF | ...
```

6.2. Parsing and type inference

We make use of the tools `ocamllex` and `ocamlyacc`. `ocamllex` is a lexical analyzer generator and transforms a set of regular expressions with token-producing actions into an appropriate OCaml source file. `ocamlyacc` is the OCaml counterpart to the famous parser generator `yacc`. `ocamlyacc` takes a context-free grammar (restricted to LALR(1)) which is annotated with semantic actions and generates a table-driven bottom-up parser as an OCaml function.

The semantic actions construct an abstract syntax tree of the parsed expression. Types are inferred by the semantic actions and added as attributes to the abstract syntax tree. For the type inference of `let` expressions, inherited attributes are required to transmit the type of the locally defined variable into the expression. To make inherited attributes available, the semantic actions are functions which map from a type environment into the set of abstract syntax trees.

6.3. Code generation

The input of the code generator is an abstract syntax tree (AST) that is annotated with type information. In this context, MetaOCaml emerges as a most useful tool, as it guides the implementor of the code generator in his task to save type information across stages. In the course of constructing code for an expression, MetaOCaml ensures that all code fragments of the subexpression, as well as the new code constructed by these fragments, have correct types. As a further convenience deriving from OCaml, MetaOCaml allows us to mix imperative with functional programming style, even in code objects. This increases the flexibility to choose both the abstraction considered appropriate for the given problem, such as a loop iterating over convolution matrices, and a good implementation in terms of execution time. Last but not least, the design and implementation of MetaOCaml is optimized to produce fast native code executables with a strong focus on efficient function applications.

6.3.1. Types for code parts

Since we must distinguish MetaOCaml code according to the type of value it generates, the return value of our code generation function is the following sum type of code for ints, floats and bools.

```
type 'a codevalue = CInt  of ('a,int)  code
                  | CBool of ('a,bool) code | ...
```

For transforming a constant value to the corresponding piece of code, we defined a lift function [5]:

```
(* val lift : value -> 'a codevalue *)
let lift = function
         | VInt   i -> CInt   .< i >.
         | VBool  b -> CBool  .< b >. | ...
```

6.3.2. Binary operations

Function `binopfunCode` takes two pieces of code as arguments. The result type does not need to be the same as the operand types, as the case for `EqI` shows.

```
  (* val binopfunCode : binOp -> 'a codevalue -> 'a codevalue
                                              -> 'a codevalue *)
  let binopfunCode op a b = match (op,a,b) with
    | (AddI, CInt a, CInt b) -> CInt   .< .~a  +  .~b >.
    | (EqI,  CInt a, CInt b) -> CBool  .< .~a  =  .~b >.
     ...
```

6.3.3. Access to image data

We provide the dimensions and the maximum pixel value at specialization time, but the individual pixel values are not available before the application of the compiled code. Thus, the contents of the image matrix is treated as a dynamic value (see Section 7). Images are represented as three-dimensional arrays, where the color value of a pixel can be addressed by a triple of row position, column position and color channel. The variables `row`, `col` and `ch` carry pieces of code for these coordinates and are combined into `pnt`, the code for getting the color value from the image. Because the image's color intensities are given as int values, a final conversion into the float interval [0.0, 1.0] is made. `source` is a structure that contains the image array in a component named `raster`.

```
  | Image ->
      let (row,col,ch) =
        match codegen_all args with
        | [CInt row; CInt col; CInt ch] -> (row,col,ch)
        | _ -> raise (Inter_error "...")
      in
      let rast = source.raster in
      let pnt = .< rast.(.~row).(.~col).(.~ch) >. in
      let maxv = float_of_int source.maxval in
      CFloat .< float_of_int .~pnt /. maxv >.
```

6.3.4. Conditional expression

The conditional expression requires a distinction of the different result types.

```
match a',b' with
 | CInt a, CInt b -> CInt .< if .~(unCBool c') then .~a else .~b >.
 | ...
```

6.3.5. Local definitions

For local definitions, we have to perform a distinction of all type combinations for the right-hand side of the definition and the result expression.

```
match (rhs.dtype, body.dtype) with
 | (Int,Float) -> CFloat .< let b = .~(unCInt rhs')
                            in .~(unCFloat (body_gen (CInt .<b>.))) >.
 | ...
```

6.3.6. Summations

Having carried out a partial or complete unrolling in the simplification phase, all the code generation has to deal with is a direct transformation of the `Sum` node into a code object representing the summation as a `for` loop. This is accomplished by the polymorphic function `dynFor`. It is parameterized by code arguments for the loop bounds (`low`, `high`), a value `init` to initialize the local reference `s` for accumulating intermediate results, and a function `step`. The function `step` generates a further addition operation where the summand is generated by function `bodycode`, which extends the environment by the iteration variable `i` and recursively applies the code generation functions for expressions.

```
...
 let dynFor low high init step =
        .< let s = ref .~init in
           for i = .~low to .~high do
              s := .~(step .<i>. .<!s>.)
           done;
           !s
        >.
 in
 begin match dtype with
 | Int ->
     let step i s = .< .~s +  .~(unCInt (bodycode i)) >. in
     CInt (dynFor lb' ub' .<0>. step)
 | ...
```

Since we distinguish the cases where the result type is either `int` or `float`, there are two alternative implementations of `step`, each producing an addition step for the respective type.

7. Examples and results

In our domain-specific language, the application program is a specification of an image processing task. Compilation is performed with respect to the height and width of a particular image and, if needed, a current iteration of a frame in a series of output images. The output frame is generated by applying the residual program to each pixel of the input image, therefore the coordinates of the output pixel are dynamic input data. The reader may be surprised that the input image is also dynamic data, even though it is available at compile-time. Here, we follow the strategy of spending specialization effort in order to save execution time only for calculations that are performed many times. Otherwise, the sum of compile-time and run-time would be extremely higher.

The impact of our optimization becomes obvious when we look at an image filtering task pixel by pixel: in the case of a pure interpretation, the application program has to be interpreted for each pixel of the output image, e.g. 307,200 times for an image of size 640×480. This motivates both the partial evaluation of the filter specification and the staged interpretation of the simplified syntax tree.

To demonstrate the versatility of our approach, we present three examples of typical image processing tasks: convolution, iterated temperature distribution, and wave animation. The implementations chosen are relatively straightforward and can be expected to be written by application programmers. Nevertheless, each of these implementations contains a certain number of static expressions that are eliminated by partial evaluation.

7.1. Convolution

A large number of image filters, such as blurring, sharpening and gradients, can be expressed by convolution. In the case of image processing, we use a special discrete and two-dimensional form of convolution. So the result of the convolution $J := M \star I$ of image I with a Matrix $M \in \mathbb{R}^{m \times m}$ is given by:

$$J_{(row,col)} = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} M_{(i,j)} \cdot I_{(row-i+\lfloor (m-1)/2 \rfloor, col-j+\lfloor (m-1)/2 \rfloor)}$$

for each image coordinate (row, col). Convolution can be seen as shifting M over the image I. Hereby, the value of each pixel $J_{(row,col)}$ is set only depending on neighboring pixels, where M governs the contribution of each of them.

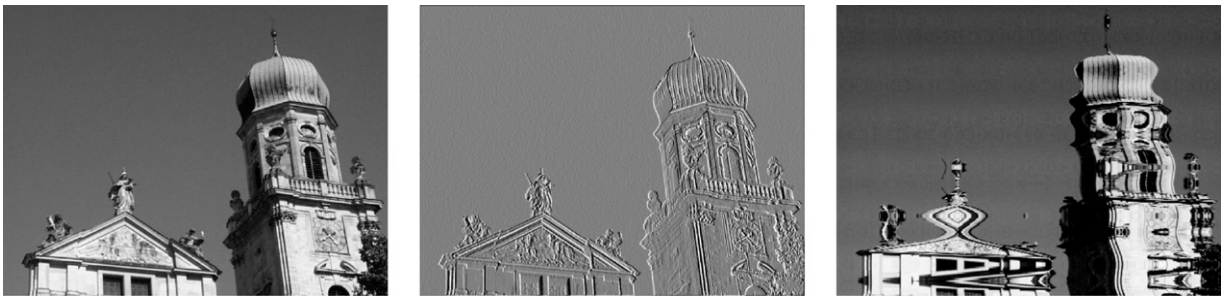

Fig. 2. (a) Original image, (b) gradient filter, (c) wave snapshot.

The index shift by $\lfloor (m-1)/2 \rfloor$ aligns the central entry of M as a coefficient for $I_{(row,col)}$. In our language defined in Section 4, we can realize a horizontal gradient filter by a convolution matrix expression of this form:

```
let m = [-1.0 0.0 1.0
        |-1.0 0.0 1.0
        |-1.0 0.0 1.0]
in
[ 3 channels:
0.5 + sum i from 0 to 2 of
        sum j from 0 to 2 of
          m[i,j] * image(row-i+1, col-j+1, current)
]
```

The effect of this filter on the original image can be seen in Fig. 2(b).

7.2. Iterated temperature distribution

Computer simulations are attractive in the design of physical systems. As an example of such a simulation, we implemented an image filter that calculates the development of a temperature distribution while a heated object moves over a plane.

This filter extends a convolution filter for blurring with a heated object, represented by a disc of radius 20 and a position that is dependent on the current iteration. In the case that a pixel lies within the object, i.e., `inobj` is evaluated to `true`, the temperature is set to 0.0, the color value which represents the highest temperature value. Otherwise, the usual convolution summation is performed.

Starting with an input image to define the initial temperature distribution, a sufficiently big number of filtering iterations, consecutively applied to each preceding result, produces an animation showing the temperature pattern as the disc moves over the plane.

```
let obr = height / 10 + iter * (height / 150.0)
        + (height / 4) * sin(iter / 10.0) in
let obc = width / 2 + (width / 4) * cos (iter / 30.0) in
let rad = 20.0 in
let inobj = (col - obc)**2 + (row - obr)**2 <= rad**2 in
let m = [ 0.0 1.0 3.0 1.0 0.0
        | 1.0 2.0 4.0 2.0 1.0
        | 3.0 4.0 6.0 4.0 3.0
        | 1.0 2.0 4.0 2.0 1.0
        | 0.0 1.0 3.0 1.0 0.0 ] in
let s = sum r from 0 to 4 of sum c from 0 to 4 of m[r,c] in
[ 3 channels:
  if inobj then 0.0
  else (sum r from -2 to 2 of
           sum c from -2 to 2 of
              image(row + r, col + c, current) * m[r + 2, c + 2]) /s
]
```

7.3. Wave animation

As seen in the previous example, simple kinds of particular animations can be implemented by the iterated application of a filter to an image and by using the iteration variable to express the progress more explicitly. In the following example, we present a filter that produces a wave-like animation of an input image.

```
let r1 = row + floor(20 * sin((iter + row) * 3.14 / 30)) in
let r2 =
  if r1 <= 0 then 0 else if r1 >= height then height-1 else r1 in
[ 3 channels: image(r2, col, current) ]
```

This filter is written to be applied independently for each animation frame, i.e., each filtering is performed on the same input image. The wave effect can be seen in Fig. 2(c) and is a result of the displacement of each row, which depends on the result of a sine function. As variable `iter` increases, the offset of the sine is shifted, producing a vertical movement of the waves. In the second `let` definition, an additional boundary correction prevents illegal image accesses beyond the bottom or top lines of the image.

7.4. Benchmark results

All run-time tests were performed on a machine with a 1GHz Pentium III processor and 512MB of main memory. For the benchmarks, we used a MetaOCaml version that provides code generators for bytecode and native code. Memory caching did not have a considerable effect on the execution times. We tested this by iterating the image processor within a loop program.

To compare the performance of different image processing tasks, the first series of tests (Table 1) was run on the example filters that we presented in the previous subsections. We examined all possible combinations of the following configurations:

- Filters:
  - Gradient filtering by convolution.
  - Temperature distribution (single iteration).
  - Wave animation (single iteration).
- Compilation: bytecode (ocamlc) or native code (ocamlopt)
- Filter execution:
  - Staged, i.e., by run-time code generation using the staged interpreter and the `.!` operator, and by running the residual code for each pixel.
  - Unstaged, i.e., by interpretation of the filter expression for each pixel. The unstaged interpreter that is used is constructed by stripping all MetaOCaml code from the code generation function (`codegen`).
- Simplification of filter expressions: (PE? yes or no)

For all timings, we used the `Trx.timenew` timing function provided by the `Trx` module of MetaOCaml [14], which reports reliable timings. Separate measurings were made for the simplification phase, for the code generation, and for the application of the `.!` operator. For each example in the benchmarks, we also provide the speed-up of the total execution time. These speed-ups are given relative to the timing of a native compilation that does not carry out a simplification and executes filter expressions by interpretation. This configuration, which relates to the common rapid prototyping of a domain-specific language, corresponds to Fig. 1 without the shaded components.

Each set of results for a given example shows that the time for simplification or code generation is marginal compared to the overall execution time. Only the execution of the `.!` operator makes a noticeable but small contribution (below 5%), which is highest when it generates native code. Examining the speed-ups, we see that the staged program is always faster.

For a complete comparison only, we also show the bytecode execution in the unstaged case. In comparison with the native code execution, which is possible without any effort of the DSL implementer, this results in a slowdown.

We remind the reader that we aim for the reuse of existing compiler technology. The speed-up of 75 due to staging in the temperature example exploits the optimization provided by the MetaOCaml compiler. In the same example, our

Table 1
Execution times on a Pentium III 1GHz 512MB, image size 1024 × 768

Filter	Compilation	Filter exec.	PE?	Time in ms					Speed-up to ∗
				PE	Codegen	. !	Execution	Total	
Gradient	Bytecode	Staged	Yes	3.39	3.22	19.26	9 369.82	9 395.68	20.61
			No		1.24	11.74	20 372.45	20 385.43	9.50
		Unstaged	Yes	3.40			238 340.00	238 343.40	0.81
			No				591 319.00	591 319.00	0.33
	Native	Staged	Yes	1.06	1.46	92.63	2 496.84	2 591.99	74.72
			No		0.52	66.35	3 936.93	4 003.80	48.37
		Unstaged	Yes	1.07			65 147.89	65 148.96	2.97
			No				193 671.90	193 671.90	∗ 1.00
	# Nodes in AST before PE: 80, after PE: 226								
Temperature	Bytecode	Staged	Yes	6.94	9.03	55.67	25 469.33	25 540.98	28.90
			No		1.92	21.38	59 875.03	59 898.33	12.32
		Unstaged	Yes	6.95			699 085.50	699 092.45	1.06
			No				1933 161.00	1933 161.00	0.38
	Native	Staged	Yes	2.66	5.44	252.64	6 324.97	6 585.70	112.09
			No		0.90	75.82	9 720.45	9 797.17	75.35
		Unstaged	Yes	2.71			226 840.80	226 843.51	3.25
			No				738 178.20	738 178.20	∗ 1.00
	# Nodes in AST before PE: 163, after PE: 613								
Wave animation	Bytecode	Staged	Yes	0.11	0.67	8.58	4 177.96	4 187.32	4.56
			No		0.73	9.19	4 217.82	4 227.74	4.52
		Unstaged	Yes	0.11			49 281.43	49 281.54	0.39
			No				64 817.72	64 817.72	0.29
	Native	Staged	Yes	0.03	0.25	39.62	1 510.48	1 550.37	12.32
			No		0.27	40.33	1 516.76	1 557.36	12.26
		Unstaged	Yes	0.03			15 217.92	15 217.95	1.26
			No				19 098.92	19 098.92	∗ 1.00
	# Nodes in AST before PE: 48, after PE: 38								

partial evaluation leads to a speed-up of about 3 in the unstaged case and still improves the staged version by a factor of about 1.5. We can see that staging and partial evaluation together can achieve speed-ups of about 112, compared to a naive DSL interpretation which already uses the OCaml native code compiler.

Comparing the high speed-ups of the temperature distribution with the lower speed-ups of the gradient filtering gives rise to the assumption that a more complex filter definition is likely to yield a better speed-up.

Another distinctive feature of our benchmarks is the effect of simplification. Gradient filtering and temperature distribution seem to gain most, whereas the wave animation is more or less indifferent to this optimization. The reason is the large portion of static expressions that are simplified. These filters have a static coefficient matrix and loops with static but small iteration ranges. While unrolling, the simplification inserts the static matrix entries and eliminates or simplifies multiplication expressions with factors such as 0 or 1. The unrolling is complete, as the extent of the body multiplied with the size of the loop index space is way below the limit of M (set to 2000).

The following object code is the result of the gradient filter specification being simplified and transformed by the code generator. We only show the first third of the code of one color channel to impart an idea of the code repetitions.

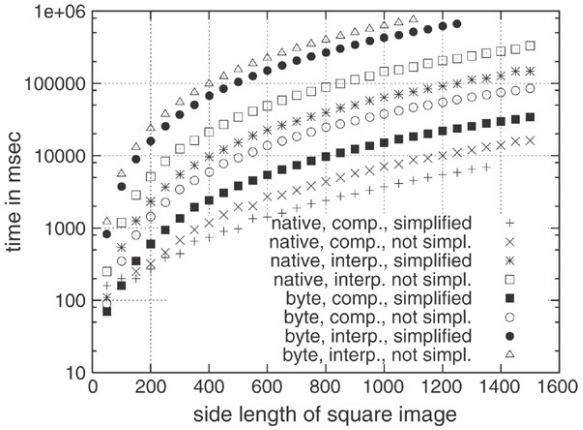

Fig. 3. Run-times of gradient filter on various image sizes.

```
.< fun (row_1, col_2) -> (Array.of_list
 [let c_3 = int_of_float
  (0.5 +.
  (((-1. *.
    (float_of_int
      ((rast.(row_1 + 1).(col_2 + 1).(0)) /. 255.)) +.
    float_of_int
      ((rast.(row_1 + 1).((col_2 - 2) + 1).(0) /. 255. +.
```

In contrast, the wave animation program barely contains static subexpressions, because most expressions depend on the dynamic index variables of the image.

The second, extensive series of experiments (Fig. 3) consisted of tests on image sizes from 50×50 to 1500×1500. Since we chose a logarithmic scale in Fig. 3, relative speed-ups appear as constant distances between markers. For images larger than about 200×200 pixels, the measured speed-ups appear more or less constant, but tend to decline for images with smaller sizes. Nevertheless, the image size where we break even with our optimization, compared with an interpreted approach, is below 50×50, which is about the size of GUI icons.

8. Conclusions

We have demonstrated that a combination of partial evaluation, algebraic simplifications, and staged interpretation in MetaOCaml permits a relatively simple implementation of a domain-specific language with significant optimizations, as our benchmarks show. The separate partial evaluation based on the abstract syntax tree avoids inspection of the generated code, which would complicate the applicability of algebraic laws. Thus, the current lack of code inspection possibilities in MetaOCaml is pointless. Although the stage annotations in the interpreter are syntactically fixed, the partial evaluation guides a performance-directed creation of appropriate pieces of code. We have shown an extreme case of this in our loop unrolling example.

Most of the speed-ups were gained from reusing existing compiler technology, especially the MetaOCaml code generator. The feasibility of this reuse in a DSL implementation is another important result of this paper. In contrast to meta-programming tools based on imperative languages like C++, or other functional languages like MetaScheme, MetaOCaml provides a high degree of safety via its type system.

Although our domain-specific language is quite small, we consider dealing with explicit binding times to be annoying for the domain programmer. Thus, we equipped our implementation with a calculation of binding times during partial evaluation to gain the necessary information for an automatic generation of a staged program. Despite the fact that language constructs can have several useful combinations of static/dynamic binding, e.g., the summation, a simple domain with two abstract values was sufficient.

Even the bytecode, if staged and partially applied, is faster than the unstaged native OCaml code, as our experiments revealed. The bytecode produced contains almost only the operations that are necessary for the computation itself. Nevertheless, the additional performance gain by compiling to native code is significant.

One of the authors investigated the possibilities of *offshoring* in the generation of OCaml programs for high-performance computing [31]. Offshoring attains C-like performance by mapping an adequate subset of the OCaml language to C [32]. In the object code restricted to this language subset, program constructs are translated into corresponding constructs in the C language. The resulting code is then compiled by a standard C compiler and dynamically linked by the invoking OCaml meta-program. The use of offshoring differs from traditional compiling in its light-weight transformation of only small program pieces and its reliance on existing compilers for generating optimized code. An application programmer can easily utilize offshoring by using the MetaOCaml brackets and escape operator and a special form of the run operator: `.!{Trx.run_gcc}`.

Acknowledgements

We would like to thank Cristiano Calcagno for installing a combination of MetaOCaml with a native code compiler for OCaml on our parallel machine. We enjoyed the discussions with Albert Cohen, Paul Kelly, Christian Lengauer and many other people. The German research foundation *Deutscher Akademischer Austauschdienst* (DAAD) provided travel support in the ARC and Procope exchange programs. Also, we would like to thank Christian Lengauer, Walid Taha and the anonymous reviewers for comments of exceptionally high value.

References

[1] http://www.infosun.fmi.uni-passau.de/cl/metaprog/index.html.
[2] http://caml.inria.fr/.
[3] http://www.cs.rice.edu/~taha/MetaOCaml/.
[4] C. Calcagno, E. Moggi, W. Taha, ML-like inference for classifiers, in: D. Schmidt (Ed.), European Symposium on Programming (ESOP), in: Lecture Notes in Computer Science, vol. 2986, Springer-Verlag, 2004, pp. 79–93.
[5] N.D. Jones, C.K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice Hall, 1993, http://www.dina.dk/~sestoft/pebook/pebook.html.
[6] N. Christensen, R. Glück, Offline partial evaluation can be as accurate as online partial evaluation, ACM Transactions on Programming Languages and Systems 26 (1) (2004) 191–220.
[7] P. Hudak, Building domain-specific embedded languages, ACM Computing Surveys 28 (4es) (1996).
[8] C. Elliott, S. Finne, O. de Moor, Compiling embedded languages, in: W. Taha (Ed.), Semantics, Applications, and Implementation of Program Generation (SAIG 2000), in: Lecture Notes in Computer Science, vol. 1924, Springer-Verlag, 2000, pp. 9–27.
[9] Y. Futamura, Partial evaluation of computation process — an approach to a compiler–compiler, Higher-Order and Symbolic Computation 12 (1999) 381–391. Originally published in Systems, Computers, Controls 2 (5) (1971) 45–50.
[10] N.D. Jones, Mix ten years later, in: Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'95), ACM Press, 1995, pp. 24–38.
[11] B. McNamara, Y. Smaragdakis, Functional programming with the FC++ library, Journal of Functional Programming 14 (4) (2004) 429–472.
[12] K. Czarnecki, J.T. O'Donnell, J. Striegnitz, W. Taha, DSL implementation in MetaOCaml, Template Haskell and C++, in: C. Lengauer, D. Batory, C. Consel, M. Odersky (Eds.), Domain-Specific Program Generation, in: Lecture Notes in Computer Science, vol. 3016, Springer-Verlag, 2004, pp. 51–72.
[13] R. Glück, J. Jørgensen, Multi-level specialization, in: J. Hatcliff, T. Mogensen, P. Thiemann (Eds.), Partial Evaluation: Practice and Theory, in: Lecture Notes in Computer Science, vol. 1706, Springer-Verlag, 1998, pp. 326–337.
[14] W. Taha, A gentle introduction to multi-stage programming, in: C. Lengauer, D. Batory, C. Consel, M. Odersky (Eds.), Domain-Specific Program Generation, in: Lecture Notes in Computer Science, vol. 3016, Springer-Verlag, 2004, pp. 30–50.
[15] W. Taha, H. Makholm, J. Hughes, Tag elimination and Jones-optimality, in: O. Danvy, A. Filinski (Eds.), Programs as Data Objects (PADO-II), in: Lecture Notes in Computer Science, vol. 2053, Springer-Verlag, 2001, pp. 257–275.
[16] C.A. Herrmann, Generating message-passing programs from abstract specifications by partial evaluation, Parallel Processing Letters 15 (3) (2005) 305–320.
[17] C. Lengauer, Program optimization in the domain of high-performance parallelism, in: C. Lengauer, D. Batory, C. Consel, M. Odersky (Eds.), Domain-Specific Program Generation, in: Lecture Notes in Computer Science, vol. 3016, Springer-Verlag, 2004, pp. 73–91.
[18] C.A. Herrmann, C. Lengauer, A transformational approach which combines size inference and program optimization, in: W. Taha (Ed.), Semantics, Applications, and Implementation of Program Generation (SAIG'01), in: Lecture Notes in Computer Science, vol. 2196, Springer-Verlag, 2001, pp. 199–218.
[19] O. Kiselyov, K.N. Swadi, W. Taha, A methodology for generating verified combinatorial circuits, in: EMSOFT'04: Proceedings of the 4th ACM International Conference on Embedded Software, Pisa, Italy, ACM Press, 2004, pp. 249–258.
[20] O. Beckmann, A. Houghton, M. Mellor, P.H. Kelly, Runtime code generation in C++ as a foundation for domain-specific optimisation, in: C. Lengauer, D. Batory, C. Consel, M. Odersky (Eds.), Domain-Specific Program Generation, in: Lecture Notes in Computer Science, vol. 3016, Springer-Verlag, 2004, pp. 291–306.
[21] Intel Corporation, Intel Image Processing Library, Reference Manual, no. 663791-004 (1997-2000).
[22] GIMP Documentation Team, GIMP User Manual, http://www.ftgimp.com/help/C/contents.html, 2002.

[23] MathWorks, Inc., Image Processing Toolbox, http://www.mathworks.com/access/helpdesk/help/toolbox/images/, 2004.
[24] P.H. Andersen, Partial evaluation applied to ray tracing, in: W. Mackens, S.M. Rump (Eds.), Software Engineering im Scientific Computing, Vieweg, 1996, pp. 78–85.
[25] B. Guenther, T. Knoblock, E. Ruf, Specializing shaders, in: Proc. Conference on Computer Graphics (SIGGRAPH-95), ACM Press, 1995, pp. 343–350.
[26] R. Glück, J. Jørgensen, Generating optimizing specializers, in: International Conference on Computer Languages, IEEE Computer Society Press, 1994, pp. 183–194.
[27] M. Sperber, P. Thiemann, Two for the price of one: Composing partial evaluation and compilation, in: SIGPLAN Conference on Programming Language Design and Implementation, 1997, pp. 215–225.
[28] S. Thibault, C. Consel, J. Lawall, R. Marlet, G. Muller, Static and dynamic program compilation by interpreter specialization, Higher-Order and Symbolic Computation 13 (3) (2000) 161–178.
[29] A. Cohen, S. Donadio, M.-J. Garzaran, C. Herrmann, O. Kiselyov, D. Padua, In search of a program generator to implement generic transformations for high-performance computing, doi:10.1016/j.scico.2005.10.013.
[30] C.A. Herrmann, T. Langhammer, Automatic staging for image processing, Tech. Rep. MIP-0410, University of Passau, Department of Informatics and Mathematics, 2004.
[31] T. Langhammer, Tuning MetaOCaml programs for high performance, Diploma Thesis, http://www.fmi.uni-passau.de/cl/arbeiten/Langhammer.pdf, 2005.
[32] J. Eckhardt, R. Kaiabachev, E. Pašalić, K. Swadi, W. Taha, Implicitly heterogeneous multi-stage programming, http://www.cs.rice.edu/~taha/publications/preprints/2005-04-16.pdf, 2005.