A Compiler for $\mathcal{HDC}$

Christoph Herrmann, Christian Lengauer,
Robert Günz, Jan Laitenberger and Christian Schaller

Fakultät für Mathematik und Informatik,
Universität Passau, Germany

{herrmann,lengauer}@fmi.uni-passau.de
http://www.fmi.uni-passau.de/~lengauer/

May 1999

Abstract

We present a compiler for the functional language $\mathcal{HDC}$, which aims at the generation of efficient code from high-level programs. $\mathcal{HDC}$, which is syntactically a subset of the widely used language Haskell, facilitates the clean integration of skeletons with a predefined efficient parallel implementation into a functional program. Skeletons are higher-order functions which represent program schemata that can be specialized by providing customizing functions as parameters. The only restriction on customizing functions is their type. Skeletons can be composed of skeletons again. With $\mathcal{HDC}$, we focus on the divide-and-conquer paradigm, which has a high potential for an efficient parallelization.

We describe the most important phases of the compiler: desugaring, elimination of higher-order functions, generation of an optimized directed acyclic graph and code generation, with a focus on the integration of skeletons. The effect of the transformations on the target code is demonstrated on the examples of polynomial product and frequent set.

Contents

1 Introduction 1

2 The Language $\mathcal{HDC}$ 3

2.1 Program structure . . . 3
2.2 Types . . . 3
2.2.1 Type expressions . . . 3
2.2.2 Type classes . . . 4
2.2.3 Type constraints . . . 4
2.2.4 User-defined types . . . 4
2.3 Function definitions . . . 5
2.4 Expressions . . . 5
2.4.1 Variables . . . 5
2.4.2 Constants . . . 5
2.4.3 Function application . . . 6
2.4.4 Infix binary operations . . . 6
2.4.5 Lambda abstractions . . . 6
2.4.6 Conditional . . . 7
2.4.7 Tuples . . . 7
2.4.8 Lists . . . 7
2.4.9 List constructor (:) . . . 7
2.4.10 Patterns . . . 8
2.4.11 `let` expressions . . . 8
2.4.12 `case` expressions . . . 8
2.4.13 Arithmetic sequences . . . 9
2.4.14 List comprehensions . . . 9

3 Skeletons 10

3.1 Skeletons for commonly used functions . . . 11
3.1.1 `map` . . . 11
3.1.2 `red` . . . 11
3.1.3 `scan` . . . 11
3.1.4 `filter` . . . 11
3.2 $\mathcal{DC}$ skeletons . . . 12
3.2.1 `dc0` . . . 12
3.2.2 `dc4io` . . . 12
3.3 Skeletons for improved efficiency . . . 14
3.3.1 `while` . . . 14
3.3.2 `sinGen` . . . 14
3.4 Evaluation control . . . 14
3.4.1 `strict` . . . 14
3.5 Input/Output . . . 14

3.5.1 `put` . . . 15
3.5.2 `get` . . . 15

4 The Structure of the Compiler **15**
4.1 Scanner and parser . . . 15
4.2 Desugaring . . . 16
4.3 List comprehension simplification . . . 16
4.4 Lambda lifting, `let` elimination . . . 18
4.5 Type checking . . . 18
4.6 Monomorphization . . . 18
4.7 Elimination of functional arguments . . . 19
4.7.1 Principles . . . 19
4.7.2 Rules . . . 21
4.7.3 Example . . . 21
4.7.4 Comments . . . 23
4.8 Elimination of mutual recursion . . . 24
4.8.1 Elimination by inlining . . . 24
4.8.2 Elimination by emulation . . . 24
4.9 `case` elimination . . . 25
4.10 Generation of intermediate DAG code . . . 25
4.11 Tuple elimination . . . 26
4.12 Optimization cycle . . . 26
4.12.1 Inline expansion . . . 27
4.12.2 Rule-based DAG optimizations . . . 28
4.12.3 Size inference . . . 28
4.13 Abstract code generation . . . 30
4.14 Space-time mapping . . . 33
4.15 Code generation . . . 34
4.15.1 DAG compilation . . . 34
4.15.2 Skeleton generation . . . 34
4.15.3 Run-time library . . . 35

5 The Parallel Run-Time Environment **35**
5.1 The model of parallel execution in $\mathcal{HDC}$ . . . 35
5.2 Organization . . . 36
5.3 Interaction with skeleton implementations . . . 36

6 Examples **37**
6.1 Karatsuba's polynomial product . . . 37
6.2 Frequent set . . . 39

7 Experimental Results 42
7.1 The effect of optimizations . . . 43
7.2 Sequential execution times . . . 44
7.3 Potential for Parallelism . . . 44

8 Using the $\mathcal{HDC}$ Compiler and Interpreter 45
8.1 Menu structure . . . 45
8.1.1 Initial menu . . . 46
8.1.2 Higher-order elimination menu . . . 46
8.1.3 Optimization menu . . . 47
8.1.4 Settings menu . . . 48
8.2 Interpreter . . . 49
8.3 Directory structure . . . 49
8.4 The prelude parts . . . 50

9 Related Work 50

10 State of the Implementation 51

A Prelude.hdc 52

B Karatsuba in Haskell 57

Acknowledgements 58

References 58

1 Introduction

Massive parallelization is an important issue when dealing with computation-intensive problems like weather forecasting, image and signal processing, system simulation or solving large systems of linear equations or inequations. The manual development of a parallel program for a specific problem may lead to efficient code but is time-consuming and error-prone, and programs developed this way are usually difficult to reuse for similar problems.

We propose a different approach in which the program does not contain explicit parallel instructions. Instead we provide a class of *skeletons*, i.e., program schemata, which we can specialize by *customizing* functions supplied as parameters, and for which we provide efficient parallel implementations. We concentrate on *divide-and-conquer* ($\mathcal{DC}$) skeletons. $\mathcal{DC}$ reduces the problem size quickly, induces a natural partitioning and contains few dependences which could limit parallelism.

We specify $\mathcal{DC}$ skeletons in a functional language, although our target language for the parallel implementations is imperative (currently C+MPI). This gives us the benefits of abstraction, which unburdens the programmer from the issues of parallelism. Also, it allows for *equational reasoning*, which helps the implementer in the derivation of a correct and efficient implementation. The name of our language is $\mathcal{HDC}$ (for $\mathcal{H}$igher-order $\mathcal{D}$ivide-and-$\mathcal{C}$onquer), see ($\mathcal{HDC}$ website, 1999). Its syntax is like that of Haskell (Bird, 1998), since Haskell possesses many of the properties we need, especially:

Strong typing. Haskell's type system provides much more support for the safe use of skeletons than the target language C.

Higher-orderness. Higher-order functions provide a powerful, general basis for the specification of skeletons. Note that customizing functions can again be skeletons.

Referential transparency. The absence of side effects permits equational reasoning, which we employ to transform skeletons.

Concise list syntax. Haskell has *list comprehensions*, which are syntactic sugar to specify lists in a closed form. The index-based list construction helps us in making the transformation from recursive Haskell skeletons to loop skeletons in C (Herrmann and Lengauer, 1997). Also, for regular sequences of integers, the $\mathcal{HDC}$ compiler retains the corresponding list comprehension with a single generator as a skeleton, named `sinGen`, in the implementation.

An added benefit is the availability of Haskell tools for the development of example programs and the possibility of a comparison of the $\mathcal{HDC}$ target code with that of Haskell compilers like the Glasgow Haskell Compiler **GHC** (Peyton Jones, 1996).

The semantics of $\mathcal{HDC}$ differs in one important point from that of Haskell: Haskell is lazy, while $\mathcal{HDC}$ is strict. The reason is that we use the parallelization technique of space-time mapping (Herrmann and Lengauer, 1996), which is based on a strict semantics. However, to increase efficiency, our compiler may choose a non-strict semantics in certain appropriate places – e.g., for the branches of a conditional, but also in some other places.

So, there is no guarantee that all arguments of a function are evaluated before the call. Skeleton implementations are hand-tailored by the skeleton implementer and, thus, are excluded from these (and any other) compiler optimizations.

The semantic difference between strictness and laziness only shows up in the termination behavior. Thus, terminating $\mathcal{HDC}$ programs deliver the same result as if evaluated under Haskell. Since we only deal with terminating programs, our $\mathcal{HDC}$ programs can be either compiled with our $\mathcal{HDC}$ compiler, or alternatively with **GHC**.

Together with the compiler, we also provide an interpreter which is able to analyze the intermediate code produced by certain phases of the compilation and report certain properties of the program to the user, like the *free schedule* (the number of steps if each operation is performed as soon as the data dependences permit), the average *degree of parallelism* (the number of parallel processors required by the free schedule), etc. Compiler and interpreter are either controlled interactively using the menu described in Sect. 8.1, or by running a Haskell script which must be compiled together with the rest of the system.

Unlike **GHC**, our compiler does not implement higher-orderness via graph reduction. Instead, each higher-order function is replaced by the collection of its first-order specializations. Functional arguments are encoded by algebraic data types which contain the function identifier and the environment of the function passed, using a modification of the *defunctionalization* method presented by Bell et al. (1997).

One might ask why we did not choose the obvious route of extending the Haskell syntax with annotations for skeleton calls and making use of one of the existing, powerful Haskell compilers. Then, the skeleton implementations (also, e.g., in C+MPI) would take Haskell closures as arguments and pass them to a new Haskell run-time environment. We decided against this for the following reasons:

- List operations in $\mathcal{HDC}$ are subject to parallelization, but a Haskell compiler constructs lists sequentially using the list constructor `(:)`. Providing a new abstract data type, which represents parallel lists, would clutter up the syntax. Also, our skeletons would have to be expressed in terms of the new data type.

- As far as we know, there exists, at present, no interface for passing Haskell closures between heaps on different processors of a distributed-memory machine. The solution of Glasgow Parallel Haskell (Trinder et al., 1998) to provide a global heap contradicts our principle of *communication-closed blocks* (see Sect. 5.1).

- Our target language, C+MPI, is implemented on a wider range of systems than the special libraries which the Haskell compilers require.

We use the following font conventions:

- **Syntactic level:** We use typewriter font, e.g., for the name `Bool`, the constructor `False` or the number constant `12`.

- **Semantic level:** Objects at this level, like the type *Bool* or the type constant *False*, are slanted. Unless otherwise specified, the semantics of a syntactic constant is defined by default by keeping the name and changing the font, and vice versa. Type arrows are written $\rightarrow$. *Lambda abstractions* are written $\lambda x.v$. Objects at the semantic level which can be identified with their counterparts at the meta level, like numbers, mathematical operators and parentheses, are not slanted.

- **Meta level:** Like this text, elements of the language of our description are written in roman font and are underlined if they belong to an algorithmic or logic language, like <u>for</u> <u>each</u> or <u>iff</u>. Syntactic equality is denoted by $\equiv$, semantic equality is by $=$. Meta variables which represent types on both the syntactic and the semantic level are written in Greek letters, like α. They are different from so-called *type variables*, which are elements of the other levels. Variables representing infix operators are written with symbols, like $\diamond$. All other variables, like x, are written in Latin letters with italic font.

2 The Language $\mathcal{HDC}$

Although $\mathcal{HDC}$ is almost a restriction of Haskell, there are some language constructs which are treated differently in the implementation.

2.1 Program structure

A program consists of a set of data type definitions (Sect. 2.2.4) and a set of function definitions (Sect. 2.3). The main function, which describes the entire input/output behavior of the program, must be named `parmain`.

2.2 Types

Like ML (Paulson, 1996) and Haskell (Bird, 1998), $\mathcal{HDC}$ has the Hindley-Milner type system (Damas and Milner, 1982). Type variables are universally quantified, i.e., a polymorphic parameter cannot be instantiated with two different types.

2.2.1 Type expressions

The language of type expressions is defined inductively by the following cases, assuming α_i are already in the language:

- `Unit` (the type which contains only one element: `Unit`)
- `Bool` (the truth values `False` and `True`)
- `Int` (restricted integers -2^{-31} to $+2^{31}-1$)
- `Double` (64-bit floating-point values)

- α_0 `->` α_1 (functions with domain type α_0 and codomain type α_1);
  `->` associates to the right; parentheses `(` and `)` can be used to group function types

- `[`α_0`]` (lists with elements of type α_0)

- `(`α_0`,`...`,`α_{n-1}`)` (n-tuples, $n > 1$)

- `IO` α_0 (input/output actions which deliver an element of type α_0)

- *tcname* $\alpha_0 ... \alpha_{n-1}$ (algebraic data types with n type arguments; see Sect. 2.2.4)

2.2.2 Type classes

In order to avoid duplication of the source code due to overloading, $\mathcal{HDC}$ contains two type classes. The type class *Num* contains the types *Double* and *Int* and is used in the definition of numerical functions. The other type class, *Ord*, contains *Double*, *Int* and *Bool* and is used for comparison. Type variables in a type expression can be restricted to a type class by replacing the type expression α with c_0 `=>` α or with `(`c_0`,`...`,`c_{n-1}`)` `=>` α, where the c_i are `Num` β or `Ord` β, where β is a type variable occurring in α.

2.2.3 Type constraints

The type of an expression is derived by successively matching two types against each other in a system of type equations. How this matching has to work is defined by type derivation rules or constraints for each language construct. Take the infix operator `==`, which compares two expressions for equality. If in the type inference the expression a `==` b is encountered and α is the type of expression a and β the type of expression b, the constraint $\{\alpha = \beta\}$ is added with consequences for the types γ_i in the subexpressions of a and b and at all other places where the γ_i are used.

2.2.4 User-defined types

The user can define additional, so-called *algebraic data types* according to the following syntax:

`data` *tcname* $\alpha_0 ... \alpha_n$ `=` $C_0\ \beta_{(0,0)} ... \beta_{(0,l(0))}$ `|` ... `|` $C_m\ \beta_{(m,0)} ... \beta_{(m,l(m))}$

tcname is the name of the type constructor that is being defined. It is parametrized with the types α_0,...,α_n. An element of the defined type can be of any of the following alternatives, separated by `|`. The actual alternative is determined by the constructor, which is one of C_0,...,C_m. Each constructor C_i is followed by a sequence of elements, which must be of the types $\beta_{(i,0)}$,...,$\beta_{(i,l(i))}$. A constructor can be viewed as a function which takes the elements of the respective types and delivers an element of the algebraic data type.

Algebraic data types provide flexibility for irregular data structures like trees. Take the following example of a binary tree defined on elements of type `a`:

```
data Tree a = Leaf a | InnerNode (Tree a) (Tree a)
```

This definition has the constructors `Leaf`, which indicates a leaf node of the tree, and `InnerNode`, which stands for a tree composed of two subtrees.

2.3 Function definitions

A function named, say, f is defined by its type, say, α and a defining equation. The type definition is given by

$f :: \alpha$

and the defining equation is given in terms of an expression e containing the free variables $x_0 \ldots x_n$:

$f\ x_0 \ldots x_n\ \texttt{=}\ e$

This definition is syntactic sugar for the following defining equation, which makes use of a lambda abstraction (Sect. 2.4.5):

$f\ \texttt{=}\ \texttt{\\}x_0\ \texttt{->}\ (\texttt{\\}x_1\ \texttt{->}\ \ldots(\texttt{\\}x_n\ \texttt{->}\ e)\ldots)$

If each x_i is of type α_i, then the type of f can be defined by:

$f\ \texttt{::}\ \ \alpha_0\ \texttt{->}\ (\alpha_1\ \texttt{->}\ \ldots(\alpha_n\ \texttt{->}\ \beta)\ldots)$

In both the lambda and the type expression the parentheses can be omitted.

Functions can be defined recursively, also indirectly (or mutually) recursively, e.g., function f depends on g and g depends on f. However, for better efficiency, recursive definitions should be avoided, predefined combinators should be used instead.

2.4 Expressions

2.4.1 Variables

A variable is a name beginning with a lowercase letter and containing only letters and digits. Internal names (also in the prelude) can also contain underscores. Each variable is associated with a static type, possibly polymorphic or restricted to a type class. $\mathcal{HDC}$ is lexically scoped, i.e., a free variable is bound to the innermost of all surrounding definitions of this variable in the program text.

2.4.2 Constants

Predefined constants are `Unit`, `False`, `True`, `[]`, integer and floating point (double) constants (Fig. 1). Constants can be defined by the user in a value definition (function definition without arguments) or as constructors of an algebraic data type, e.g.:

```
data Color = Yellow | Green | Blue
```

Type(s)	Examples
`Unit`	`Unit`
`Bool`	`False`, `True`
`Int`, `Double` (`Num`)	`0`, `-5`, `32`
`Double`	`2.3`
`[`α`]`	`[]`

Table 1: $\mathcal{HDC}$ constants

2.4.3 Function application

Function application is denoted by juxtaposition, e.g., an application of function f to an argument x is written $f\,x$. Formally, there are only functions with a single argument. A function with multiple arguments is represented as a function which takes the first argument and returns a function which is applied to the remaining arguments. This is known as *currying*. If not all arguments are given, we speak of a *partial application*. Function application associates to the left and binds more tightly than any other binary operation. Function application adds the following constraints to the set of types:
$\{f :: \alpha \rightarrow \beta,\, x :: \alpha,\, f\,x :: \beta\}$.

2.4.4 Infix binary operations

An infix operator, say $\diamond$, is syntactic sugar for a function named $(\diamond)$ taking two arguments. $\mathcal{HDC}$ borrows the sectioning mechanism from Haskell, in which $(x\diamond)$ is $(\diamond)$ with a fixed first argument x, $(\diamond x)$ is $(\diamond)$ with a fixed second argument x and $(x \diamond y)$ is $(\diamond)$ with both arguments fixed. Any function f taking two curried arguments can be used as a binary infix operator by writing `` `f` ``. For the actual operators, have a look at the prelude in Appendix A.

2.4.5 Lambda abstractions

A lambda abstraction `\`x `->` e defines a function which takes a value x and delivers the value of the expression e in which each occurrence of the free variable x has been replaced by the argument of the function:

$$(\backslash x \texttt{ -> } e)\; y \;=\; e[x := y]$$

$e[x := y]$ denotes the substitution of every free occurrence of x in e by y.
Type constraint: $\{\; x :: \alpha,\; e :: \beta,\; (\backslash x \texttt{ -> } e) :: \alpha\rightarrow\beta \;\}$.

Curried function definitions can be abbreviated by writing all arguments successively, i.e., instead of $\backslash x_0$ `->` $(\backslash x_1$ `->` $\ldots(\backslash x_{n-1}$ `->` $e)\ldots)$ one can write $\backslash x_0\, x_1 \ldots x_{n-1}$ `->` e. Also, structured arguments, so-called *patterns* (Sect. 2.4.10), like tuples or lists can be used. See the following examples:

```
\x -> x+1
\x y -> x+y        instead of  \x -> \y -> x+y
\(x,y) -> x+y      instead of  \z -> fst z + snd z
\[x,y] -> x+y      instead of  \z -> z!!0 + z!!1
```

The first definition describes a function which returns its argument incremented. The second takes a value `x` and delivers a function which takes a value `y` and returns the sum of `x` and `y`. The other two examples take a pair resp. a list and deliver the sum of both components.

2.4.6 Conditional

The syntax of a conditional is

`if` *cond* `then` *t* `else` *e*

This expression is strict in *cond*, but not strict in t and e. If *cond* evaluates to *True*, t is evaluated and returned as the value of the conditional. Otherwise, e is evaluated and returned.

Type constraint: { *cond* :: *Bool*, t :: α, e :: α, (`if` *cond* `then` t `else` e) :: α }.

Example: (factorial function)

```
fac n = if n==0 then 1 else n * fac (n-1)
```

2.4.7 Tuples

A tuple is an ordered, fixed-size collection of components $x_0 ... x_{n-1}$, denoted by (x_0,...,x_{n-1}), where $n > 1$. If $n = 2$, we speak of a *pair*. The components need not be of the same type. The elements of a tuple can be selected by pattern matching (Sect. 2.4.10).

2.4.8 Lists

A list is an ordered, arbitrary-size collection of components of the same type. A list of length n can be given explicitly by [x_0,...,x_{n-1}].

2.4.9 List constructor (:)

The constructor (:) takes an element x_0 of type α and a list [x_1,...,x_{n-1}] of type $[\alpha]$ and delivers the list [x_0,...,x_{n-1}] of type $[\alpha]$. Contrary to Haskell, in $\mathcal{HDC}$, applying a single : or `tail` is expensive, i.e., linear in the length of the list (in Haskell it is constant). The advantage of $\mathcal{HDC}$ is that the linear chain of dependences present in Haskell lists does not exist in $\mathcal{HDC}$ lists. This allows for a constant-time element access via the $\mathcal{HDC}$ version of the index function (!!). The philosophy followed here is to exploit the $\mathcal{DC}$ paradigm and, thus, the assumption is that a list is constructed by appending two (or more) lists of roughly the same length. Alternatively, one can use list comprehensions (Sect. 2.4.14) to compute all elements of a list simultaneously.

2.4.10 Patterns

Patterns are expressions which consist only of constructors (including the tuple and list constructor), constants and disjoint names. A pattern can occur as an argument in a function definition or lambda abstraction, or on the left-hand side of a `case` branch (Sect. 2.4.12) or `let` expression (Sect. 2.4.11). In this case, the pattern of the formal parameter is matched against the value passed as actual parameter. A match of a name always succeeds, with the consequence that the variable associated with this name is bound to the value. A match of a composite expression succeeds if the constructors are identical and the matches of all corresponding components succeed. Then, the environment is constructed by accumulating the bindings of all component matches. In the case of lists and tuples, both sides must have the same number of components, because the constructors of tuples of different sizes are distinct and list structures are desugared into a sequence of the binary `:` constructor. If a pattern match fails, the next alternative in a list of defining equations or case branches is tried. If no alternative remains, a run-time error occurs.

2.4.11 `let` expressions

`let` expressions are used for defining a local environment of values and functions, which can be (mutually) recursive. The form of a `let` expression is:

`let` $\{\ eq_0$; ... ; $eq_n\ \}$ `in` e

where e is the expression which forms the value returned and the eq_i are equations of the form pat_i = e_i, where pat_i is a pattern and e_i an expression. If a pattern is an application of a variable ($pat_i \equiv f_i\ x_0\ ...\ x_{m(i)}$) the equation defines a local function with function symbol f_i. `let` expressions are desugared by a process called *lambda-lifting* (Johnsson, 1985; Peyton Jones, 1987). If it should turn out in the long run that the program optimizations at later phases cannot identify the common subexpressions originating from the elimination of `let` expressions, other transformations will have to be considered.

A simplified *layout style* of Haskell can be used here as well as for the branches of `case` expressions: braces and semicolons can be omitted if all eq_i have the same indentation, which is larger than the indentation of the `let`.

2.4.12 `case` expressions

A case expression defines a value by case distinction. The form of a `case` expression is:

`case` sel `of` $\{\ branch_0$; ... ; $branch_n\ \}$

where $branch_i \equiv pat_i$ `->` exp_i. sel defines the value used for the case analysis. Each pat_i is a pattern to be matched with the value of sel, exp_i delivers the result of the `case` expression if the ith branch is the first whose pattern pat_i matches (Sect. 2.4.10). As an example, here is a function which sums up the numbers at all leaves of an instance of the binary tree defined in Sect. 2.2.4.

```
sumup :: Num a => Tree a -> a
sumup tree = case tree of
               Leaf x -> x
               InnerNode leftSub rightSub -> sumup leftSub + sumup rightSub
```

As in `let` expressions, the pattern on the left side defines bindings for variables (here: `x` resp. `leftSub` and `rightSub`) which can be used on the right side. The layout style, which lets us define this expression without the use of braces and semicolons, requires that `Leaf` and `InnerNode` have the same indentation which is larger than the indentation of `case`.

2.4.13 Arithmetic sequences

The arithmetic sequence, denoted `[`a`..`b`]`, produces a list of integers ranging from a to b.

Example: `[1..6]` = `[1,2,3,4,5,6]`

2.4.14 List comprehensions

A list comprehension is a convenient Haskell construct for defining lists. The syntax of the list comprehension is:

`[` e `|` q_0 `,`$\dots$`,` q_{n-1} `]`

$q_0, \dots, q_{n-1}$ is a sequence of *qualifiers* which produce a list *el* of environments, which has the same length as the result of the list comprehension. The ith element of the result is obtained by evaluating expression e in the ith element of the environment list. The list of qualifiers is traversed from left to right. The initial *el* has length 1 and contains no bindings. A qualifier can be either a *generator* or a *guard*.

- A generator (i `<-` xs) refines the *el* as follows. Each element, say, *env* of *el* is taken as an environment for evaluating xs to a list of length l. *env* is removed from *el* and replaced by l new entries, one for each element of xs. The jth new entry contains the old environment *env* plus a binding for i to the jth element of xs. i must be a variable.
- A guard is an expression of type *Bool* which, if evaluated to *True*, keeps *el* unchanged and otherwise deletes the current element from *el*.

A formal semantics of list comprehensions is defined in Sect. 4.3.

List comprehensions correspond to loop nests in imperative languages. The ith qualifier is located at the ith level of nesting. A generator corresponds to a loop and a filter to a condition which governs the execution of the enclosed nest. This correspondence is exploited in the implementation of skeletons.

Examples:

```
[  i+1   | i<-[0,1,2] ]             = [1,2,3]
[ (i,j)  | i<-[0,1,2], j<-[0..i] ]  = [(0,0),(1,0),(1,1),(2,0),(2,1),(2,2)]
[   i    | i<-[0,1,2], even i ]     = [0,2]
```

3 Skeletons

A skeleton is a predefined program schema. (The imperative world would call it a "template".) If it corresponds to a particular class of algorithms, like $\mathcal{DC}$, it is called an *algorithmic skeleton*. If it corresponds to a class of machine operations, it is called an *architectural skeleton*. An example is the *broadcast* operation, which sends a message from one processor to all other processors. Architectural skeletons are only of interest in the code generation for a particular target machine or message-passing library. In the following, we say just *skeleton* when we mean an algorithmic skeleton.

Skeletons have been used widely for parallel programming (Cole, 1989; Darlington et al., 1993; Busvine, 1993; Bratvold, 1994; Darlington et al., 1995; Bacci et al., 1995; Botorog and Kuchen, 1996; Gorlatch, 1996; Ciarpaglini et al., 1997; Gorlatch and Pelagatti, 1999). $\mathcal{HDC}$ lends special support to programming with skeletons. The $\mathcal{DC}$ strategy can be expressed formally as a skeleton which is instantiated with problem-specific, customizing functions. E.g., the *mergesort* algorithm requires a customizing function which merges two ordered lists. For a sound treatment of skeletons, $\mathcal{HDC}$ provides higher-order functions. Thus, one can define a skeleton as a (recursive) function and replace it later by a predefined efficient parallel implementation.

Though a single $\mathcal{DC}$ skeleton would be sufficient for a definition of the paradigm itself, as is `dc0` defined in Sect. 3.2.1, it would not adequately reflect the variety in the structure of different $\mathcal{DC}$ algorithms. As a consequence, the use of the skeleton `dc0` for all $\mathcal{DC}$ problems would lead in many cases to bad performance of the implementation.

In order to exploit the specific structure of a $\mathcal{DC}$ algorithm, the $\mathcal{DC}$ paradigm can be refined into different specialized forms (different skeletons) with varying patterns of data dependence and data distribution (Herrmann and Lengauer, 1999).

Our special interest lies in sophisticated $\mathcal{DC}$ skeletons; a hierarchy of five such skeletons, which we call `dc0` to `dc4`, is described in Herrmann and Lengauer (1997). In the present report, `dc4io` (Sect. 3.2.2), an improved form of `dc4`, appears in our polynomial product example. The frequent set example is too complicated to fit a single skeleton. It requires many applications of simpler skeletons, which appear in this report. Some of them, like `map` or `filter`, could, in principle, be expressed by $\mathcal{DC}$.

The approach of the $\mathcal{HDC}$ compiler is to replace expensive patterns of computation by efficient predefined implementations. The mechanism for implementing a skeleton should be as easy as possible, because the need for further skeleton implementations may arise. To implement a new skeleton, the prelude has to be extended by its type definition, and the Haskell source file `Skeletons.hs` has to be extended with a function which generates the skeleton implementation. The name of an interface for the skeleton is prefixed with `skel_`, in order to recognize it as such and protect it against elimination and optimization by the compiler. However, these prefixed functions should not be used outside the prelude, because their type can differ from the Haskell type or change in future versions. The corresponding functions to be used in application programs usually have the same name, but without the prefix.

The following subsections list the skeletons which are implemented at present. For each skeleton we provide the signature, an algorithmic definition in $\mathcal{HDC}$ and an example application.

3.1 Skeletons for commonly used functions

3.1.1 `map`

Applies a function to all elements of a list.

```
map :: (a->b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

map (+1) [0,1,2] = [1,2,3]
```

3.1.2 `red`

Uses an associative function `f` to reduce a list of values to a single value.

```
red :: (a->a->a) -> a -> [a] -> a
red f n []     = n
red f n (x:xs) = f x (red f n xs)

red (+) 0 [1,2,3] = 6
```

3.1.3 `scan`

Applies `red` to all prefixes of the given list.

```
scan :: (a->a->a) -> a -> [a] -> [a]
scan f n xs = map (\i -> red f n (take i xs)) [0..length xs]

scan (+) 0 [1,2,3] = [0,1,3,6]
```

3.1.4 `filter`

Filters all elements that fulfill a predicate.

```
filter :: (a->Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs) = let rest = filter p xs
                  in if p x then x : rest
                            else     rest

filter (>2) [0,5,3,1,5] = [5,3,5]
```

3.2 $\mathcal{DC}$ skeletons

3.2.1 `dc0`

$\mathcal{DC}$ in its general form.

```
dc0 :: (a->Bool) -> (a->b) -> (a->[a]) -> (a->[b]->b) -> a -> b
dc0 p b d c x = if p x
                   then b x
                   else c x (map (dc0 p b d c) (d x))
```

If the predicate function `p` determines that the problem `x` can be trivially solved, the basic function `b` is applied. Otherwise the problem is divided by `d`, producing a list of subproblems. The algorithm is mapped recursively onto the subproblems. At last, the combine function `c` uses the input data `x` and the solutions of the subproblems to compute the solution of the original problem.

A functional version of the *quicksort* algorithm can be expressed in terms of `dc0`:

```
quicksort :: Ord a => [a] -> [a]
quicksort xs
 = let d (p:ps)         = [filter (<p) ps, filter (>p) ps]
       c (p:ps) [le,gr] = le ++ p : (filter (==p) ps ++ gr)
   in dc0 ((<2).length) id d c xs
```

`p` is the name of the *pivot*. `d` generates two subproblems of the elements that are less resp. greater than the pivot. `le` resp. `gr` are the solutions of these subproblems. `c` combines them and inserts the elements which equal the pivot in the middle.

3.2.2 `dc4io`

A special kind of $\mathcal{DC}$ which requires elementwise divide and combine operations on subblocks of data.

```
dc4io :: Int->Int->Int->(a->b)->([a]->[a])->([b]->[b])->Int->[a]->[b]
```

The definition of `dc4io` is more involved than the others, and we can only sketch it here:

```
dc4io probdegree indegree outdegree basic divide combine levels xs = ...
```

The parameters of `dc4io` have the following meaning:

- `probdegree::Int`: the degree of problem division, i.e., the number of subproblems which are generated for each problem not trivially solved; this degree is fixed in `dc4io` in contrast to `dc0`.
- `indegree::Int`: the degree of division of input data; it tells in how many blocks the input data is to be divided.

problem	probdegree	indegree	outdegree
FFT, bitonic merge	2	2	2
polynomial product	4 (3)	2	2
matrix product	8 (7)	4	4

Table 2: example $\mathcal{DC}$ division degrees

- `outdegree::Int`: the degree of composition of output data; it tells of how many blocks the output data is to be composed.
- `basic::(a->b)`: the function to be applied in the trivial case.
- `divide::([a]->[a])`: the function `divide` takes a list of length `indegree` as input and delivers a list of length `probdegree` as output; it describes how the element-wise operation computes for each particular subproblem the element i using the ith element from each input block.
- `combine::([b]->[b])`: the function `combine` takes a list of length `probdegree` as input and delivers a list of length `outdegree` as output; it describes how the element-wise operation computes for each particular output block the element i using the ith element from each subproblem solution.
- `levels::Int`: the number of recursive levels in which the $\mathcal{DC}$ tree unfolds, in contrast to `dc0` there exists no predicate for determining the trivial case; the $\mathcal{DC}$ tree is balanced and the number of levels can be computed easily from the problem size.
- `xs::[a]`: the input data; it is a list on which the division into blocks apply; likewise the output data is also of list type (`::[b]`).

`dc4io` works well for vector and matrix algorithms like FFT, bitonic merge, polynomial product and matrix product (Herrmann and Lengauer, 1997). The Karatsuba polynomial product is discussed in detail in Sect. 6.1.

`indegree` and `outdegree` depend much on the data representation, e.g., for vectors they have the value 2 (left and right part), for matrices they have the value 4 (upper left part, upper right part, lower left part and lower right part), see Tab. 2.

The parenthesized values are for the optimized version of the respective algorithm, e.g., for Karatsuba's polynomial product and Strassen's matrix product.

3.3 Skeletons for improved efficiency

3.3.1 `while`

Takes a predicate `p`, a function `f` and a value `x` and iterates `f`, starting from `x`, as long as the predicate on the input for `f` is *True*. The `while` skeleton is intended to be used instead of tail recursion in order to avoid the recursion stack.

```
while :: (a->Bool) -> (a->a) -> a -> a
while p f x = if p x
                 then while p f (f x)
                 else x
```

```
while (\(i,s) -> i<3) (\(i,s) -> (i+1,s+i*i)) (0,0) = (3,5)
```

3.3.2 `sinGen`

Takes a function `f` and a value `n` and generates a list of length `n` whose value at position `i` is computed by applying `f` to `i`. The aim of `sinGen` is to have a short representation for large, regular index sets, e.g., the odd numbers from 1 to 1000001. To make this work, `sinGen` has to be fused in program optimization (see Sect. 4.12.2).

```
sinGen :: (Int->a) -> Int -> [a]
sinGen f n = map f [0..n-1]
```

```
sinGen (\i -> i*i) 4 = [0,1,4,9]
```

3.4 Evaluation control

3.4.1 `strict`

Takes a function `f` and an argument `x` and it guarantees to evaluate `x` before calling `f`. Program optimizations will not touch the application of `f` to `x`. This skeleton is necessary to protect the `IO` monad against elimination via inlining, due to a lack of data dependences.

```
strict :: (a->b) -> a -> b
strict f x = f x
```

```
strict (+1) 1 = 2
```

3.5 Input/Output

For the input and output function, we cannot provide a purely functional definition. The reason is their interaction with the input and output streams, which are hidden in the `IO` monad. The C implementations of the skeletons are far too long to be given here – they have to deal with nested lists and tuples.

3.5.1 `put`

Takes a value and delivers an I/O action which returns `Unit`. In the I/O action, the value is appended to the standard $\mathcal{HDC}$ output channel. Printable values are of type *Int*, *Double* and also tuples and lists composed of printable values.

```
put :: a -> IO Unit
```

3.5.2 `get`

Performs an I/O action in which a value of type `a` is read from the standard $\mathcal{HDC}$ input channel. The set of readable values is the same as the set of printable values (see `put`).

```
get :: IO a
```

4 The Structure of the Compiler

The $\mathcal{HDC}$ compiler translates a subset of Haskell into an imperative language – at present, C with MPI calls. The main difference to Haskell is that $\mathcal{HDC}$ is strict, in order to facilitate a compile-time parallelization. Two implementational differences to a typical Haskell compiler are that (1) higher-order functions without a skeleton implementation are eliminated and (2) list comprehensions are simplified to a combination of (parallel) skeletons. The reason is that higher-order functions complicate but list comprehensions simplify a static space-time mapping.

The compiler is based on the principle of compilation by transformation, which has already been used successfully in **GHC**, and consists of a number of phases described in Sect. 4.1 to 4.15. An interpreter, which can be used to analyze the program with respect to correctness, performance and code structure after individual compilation phases, is presented briefly in Sect. 8.2.

4.1 Scanner and parser

The source text is translated into a set of syntax trees, one for each function in the program. Each syntax tree is represented as an algebraic data type in Haskell. First, the source code is transformed by the scanner into a token stream. This common technique in compiler construction (Aho et al., 1986) simplifies the generation of a grammar for $\mathcal{HDC}$. For an efficient parser generation, we use the parser generator *happy*, the functional equivalent of *yacc*. The parser created by *happy* generates a syntax tree which is represented by an algebraic data type in Haskell.

The layout style of Haskell is supported, i.e., indentation can be used instead of braces and semicolons to group together items at the same level of particular syntactic structures. The user can declare new operators just as, e.g., the operator `(->>)` is declared in the program for the Karatsuba example on page 38, and state their precedence and associativity (see the second line of the `karatsuba` program). This information is exploited by the parser.

```
case exp of
  Left (Right x) -> exp1
  _              -> exp2
```

$$\stackrel{\rightarrow}{=}$$

```
case exp of
  Left y -> case y of
              Right x -> exp1
              dummy1  -> exp2
  dummy0 -> exp2
```

Figure 1: Example transformation of nested patterns

4.2 Desugaring

In this phase, complex syntactic structures are translated to compositions of simpler structures. Nested patterns are eliminated, in order to simplify the code structure for the following phases. After the transformation, the pattern is either a simple variable or a constructor followed by n variables, where n is the arity of the constructor given by the definition, e.g., via `data`. Fig. 1 contains a simple transformation. An equational transformation of a into b is denoted by $a \stackrel{\rightarrow}{=} b$.

Pattern bindings like `let Left (Right x) = exp` receive a similar treatment. In this case `x` is bound to:

```
case exp of
  Left y -> case y of
              Right z -> z
              dummy1  -> error "mismatch"
  dummy0 -> error "mismatch"
```

Sometimes the flattening of a pattern can lead to cascades of `case` expressions, which blow up the size of the right-hand side of a pattern definition. This occurs when a pattern includes a constructor with several arguments and many of these arguments represent a subpattern which must also be flattened. But, in most cases, the growth of the `case` expression is not dramatic and does not affect the performance of the remaining compiler phases.

4.3 List comprehension simplification

GHC resolves comprehensions completely, up to the construction by the empty list (`[]`) and cons (`:`), usually by traversing the list of qualifiers from left to right (Peyton Jones, 1987; Haskell 98–Report, 1999). Our goal is to base list comprehensions on (parallel) skeletons. As presented here, our rewrite rules in Fig. 2 specify the traversal of the list of qualifiers in the opposite order: from right to left. This has two advantages: (1) nested `map`s are not intertwined with nested `concat`s, which preserves structural information; (2) an efficient `filter` skeleton is used instead of generating lots of empty lists in cases in which guards fail. The disadvantage is that the rules will become far more complicated if extended to the full capability of Haskell.

lcEmpty		`[` e `| ]`
	$\stackrel{\rightarrow}{=}$	`[` e `]`
lcSinGuard		`[` e `|` g `]`
$\{g$ is a guard$\}$	$\stackrel{\rightarrow}{=}$	`if` g `then [` e `] else []`
lcOptGuard		
$\{g_i$ are guards,		`[` e `|` $q_0, \ldots, q_n, x$ `<-` $xs, g_0, \ldots, g_k, \ldots, g_m$ `]`
$x \notin \mathit{freevars}(g_k)\}$	$\stackrel{\rightarrow}{=}$	`[` e `|` $q_0, \ldots, q_n, g_k, x$ `<-` $xs, g_0, \ldots, g_{k-1}, g_{k+1}, \ldots, g_m$ `]`
lcXGen		`[` e `|` $q_0, \ldots, q_n, x$ `<-` xs `]`
	$\stackrel{\rightarrow}{=}$	`concat [map (\`x `->` e`)` xs `|` $q_0, \ldots, q_n$ `]`
lcGenGuard		`[` e `|` $q_0, \ldots, q_n, x$ `<-` xs, g `]`
$\{g$ is a guard$\}$	$\stackrel{\rightarrow}{=}$	`concat [map (\`x `->` e`) (filter (\`x `->` g`)` xs`) |` $q_0, \ldots, q_n$ `]`
lcTwoGuards		`[` e `|` $q_0, \ldots, q_n, g_0, g_1$ `]`
$\{g_0, g_1$ are guards$\}$	$\stackrel{\rightarrow}{=}$	`[` e `|` $q_0, \ldots, q_n$`, if` g_0 `then` g_1 `else False ]`

Figure 2: Simplification of list comprehensions

The rewrite rules shown in Fig. 2 cover all possible list comprehension formats in our restricted language. They replace a list comprehension by applications of the skeletons `concat`, `map` and `filter` which are supposed to have efficient implementations. The rules are applied until no further application is possible. If there is a choice between several rules, the one highest up in Fig. 2 is most efficient. Rule **lcEmpty** deals with the case that the sequence of qualifiers has become empty by the other transformations.

Rule **lcSinGuard** simplifies a qualifier list consisting of a single guard. Depending on the value of the guard, the result is a list of either length 1 or length 0.

Rule **lcOptGuard** shifts a guard as far as possible to the left, in order to avoid multiple evaluations.

Rule **lcXGen** deals with the case that the last qualifier is a generator. The other qualifiers define a list of environments. In the comprehension before simplification, the last qualifier refines each element of this list by a set of new bindings for the last generator variable x. After the transformation, this refinement is shifted to the expression on the left side of the comprehension, which has been replaced by a list, one element for each instance of the last generator in the current environment, as defined by the other qualifiers. We reuse the name x of the generator variable for the lambda expression to preserve the bindings in the transformation. Note that the left side is in the scope of the environment defined by the qualifiers on the right side. Therefore, all free variables of xs are bound to the same values as before.

If a guard appears behind a generator, rule **lcGenGuard** helps to fuse the two. It is similar to rule **lcXGen**, except that the new bindings for the last generator variable, which lead to a failure of the last guard, are eliminated from the list via the `filter` skeleton before. The previous application of rule **lcOptGuard** assures that the guard really refers to the variable bound by the generator.

If two guards appear next to each other, they can be simplified to a single guard according to rule **lcTwoGuards**.

4.4 Lambda lifting, `let` elimination

Lambda abstractions and `let` expressions are eliminated by introducing auxiliary functions (Johnsson, 1985). As mentioned in Sect. 2.4.11, other elimination methods should also be considered for better efficiency.

4.5 Type checking

The type checker is based on unification using the rules by Martelli and Montanari (1982). A simple type class system is implemented by assigning a type variable a set of possible types. The unification of two type variables then involves computing the intersection of both sets.

4.6 Monomorphization

In this phase, all type variables are eliminated and replaced by the types actually needed. This requires the duplication of each function for all concrete types which occur in the context. To ensure that all type variables are eliminated, monomorphization is started from the `IO Unit` type of the `parmain` function and propagated along the call structure, using the bindings that are imposed by the type constraints of the language constructs. If a polymorphic function is called, a copy of it with monomorphic type is added to the code and the call is redirected to this copy. Arguments like `y` in `fst (x,y) = x` keep an uninstantiated type, which is implemented by a `void` argument in C, if not even eliminated by inline expansion (Sect. 4.12.1).

Monomorphization is needed because our aim is not to translate to a high-level target language but to stay close to the machine representation of data and instructions.

Example:

Consider the following program as subject of monomorphization:

```
parmain :: IO Unit
parmain = get >>= \xs ->
          put (map id (xs::[Int]))
```

```
map :: (a->b) -> [a] -> [b]
map f xs = skel_map f xs

id :: a -> a
id x = x
```

Monomorphization delivers the following result:

```
parmain :: IO Unit
parmain =  get >>= \xs ->
           put (map_T1 id_T2 (xs::[Int]))

map_T1 :: (Int->Int) -> [Int] -> [Int]
map_T1 f xs = skel_map f xs

id_T2 :: Int -> Int
id_T2 x = x
```

4.7 Elimination of functional arguments

HOE takes a program which must be well-typed according to the Hindley-Milner rules. Also, the program must be closed, i.e., all functions cited in the program must be available to the HOE procedure for a global analysis and transformation. The result of the HOE is an equivalent first-order functional program, which is also well-typed.

4.7.1 Principles

The HOE algorithm we found (Bell et al., 1997) uses a set of seven rewrite rules for the transformation. The idea is to replace the partial applications of a function by a kind of closure. A closure contains a function identifier and the values of the free variables in the partial application.

The replacement of functional arguments by closures proceeds as follows:

- A variable of a function type is left unchanged because it represents already a closure.
- A partial application of a function (see Sect. 2.4.3) is replaced by an instance of an algebraic data type in which the function identifier is represented by a constructor. The arguments of the constructor carry the values of the free variables in the partial application. These values are taken from the context of the call.
- All locations at which a functional variable is applied are replaced by a call of an apply function constructed for the respective function type. The first argument of the apply function is the closure, the following arguments are the arguments of the encoded function. The apply function applies the original function, with the respective partial

```
etaExpand :: Function → Function
etaExpand f
    = if  f returns a function as result
        then  add new variables as needed to f
        else  f
```

Figure 3: Algorithm **etaExpand**

```
encode :: Expression → State Expression
encode expr@(f  e1...ei...en)
    = if  f is function or constructor
            ∧  expr is not a function
            ∧  ei :: α is a functional argument
        then do let
                    expr′ ≡ (f  e1...(C_ei  v1...vm)...en)
                    data  ≡ (data Data_α = C_ei β1...βm)
                    apply ≡ (apply_α :: Data_α → α
                             apply_α  c  x1...xj
                                   = case c of
                                       C_ei  v1...vm -> ei  x1...xj)
                add data and apply to current program state State
                return expr′
        else  return expr
    where v1::β1,...,vm::βm are the free variables in ei
          x1,...,xj are additional arguments for eta-expanded ei
```

Figure 4: Algorithm **encode**

```
applVar :: Expression → Expression
applVar expr@(f  e1...en)
    = if  f::α is a variable ∧ expr is not a function
        then apply_α f  e1...en
        else expr
```

Figure 5: Algorithm **applVar**

application derived by the constructor, to the argument expression in the context of the call which can use values of the closure, also encoded functions.

4.7.2 Rules

Some of the seven rules, which the original HOE algorithm (Bell et al., 1997) is based on, deal with restricting polymorphism and become obsolete in our monomorphic setting. The diploma thesis of one of the authors of this report (Schaller, 1998), describes an HOE algorithm tailored for $\mathcal{HDC}$, which uses the following set of four rules:

1. **EtaExpand** (Fig. 3). This rule expands function definitions, which return functions as *result*, with as many additional formal arguments as the function returned expects. If the result was polymorphic before monomorphization, the number of additional arguments may depend on the call. Then, applications of the expanded function include the application of the function returned and deliver a non-function result.

2. **Encode** (Fig. 4). This rule encodes a functional argument using a constructor and introduces an apply function as described above. The rule is given in a state-monadic style, taking an expression as argument and returning an expression, while having access to the current program state. Depending on the type of the functional argument, the generated constructor is added to a data type, called `Data`, parametrized with an identification of the type of the argument. This is necessary for the correct generation of skeleton instances in a later phase of the compiler.

3. **ApplVar** (Fig. 5). If, in a function application, the function is represented by a variable which is marked to carry a closure value, a temporary type inconsistency occurs during the transformation because a closure cannot be applied. This rule wraps the closure in a call to an additional apply function which takes the closure as an argument.

4. **RemoveHOTypes** (Fig. 6). To clean things up, all function types appearing in data type definitions are replaced by the algebraic data type called `Data`, which is parametrized with an identifier of the encoded type and encompasses all closures.

The algorithm starts with a phase of applications of rule 1, followed by a phase in which rules 2 and 3 are applied repeatedly in any order, and terminates with a phase of applications of rule 4. All phases perform rule applications as long as possible.

4.7.3 Example

Let us study a small example for illustration. Assume the following definition of function `map`, which applies a function `f` elementwise to a list:

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

```
removeHOTypes :: Type → Type
removeHOTypes t
    = case t of
        α@(_ → _)    -> Data_α
        D t_1 ... t_n  | D algebraic data type
                     -> D (removeHOTypes t_1 ... removeHOTypes t_n)
        _            -> t
```

Figure 6: Algorithm **removeHOTypes**

Now, assume that map is used with two different functions in the first argument:

```
map inc xs             where  inc ::  Int -> Int
map (add (5*i)) xs     where  add ::  Int -> Int -> Int
```

The difference in the signatures of the two argument functions is important here.

The first of our for rules, **EtaExpand**, does not take hold because the result of map applied to two arguments is not a function. To spare the reader the confusion of type inconsistencies, we apply rule **RemoveHOTypes** not at the end but simultaneously with rule **Encode**.

To encode the arguments inc and add, the data type T_1 and an apply function for it (rule **Encode**) are created:

```
data T_1 = C_inc | C_add5times Int

apply_T_1 :: T_1 -> Int -> Int
apply_T_1 code x  = case code of
                      C_inc         -> inc x
                      C_add5times i -> add (5*i) x
```

We have chosen intuitive names for the generated constructors. The HOE procedure generates synthetic names, of course.

Note that the constructor C_add5times has an argument i. This is because i appears as a free variable in add (5*i). The scope of a free variable in a call is the scope of the caller and, therefore, the value of the free variable must be passed.

Next, the applications above have to be replaced:

```
map_T_1 C_inc xs
map_T_1 (C_add5times i) xs
```

The call of the first argument of map in the body of map must be replaced by an apply function (rule **ApplVar**). Thus, map is transformed as follows:

```
map_T_1 :: T_1 -> [Int] -> [Int]
map_T_1 fcode []     = []
map_T_1 fcode (x:xs) = apply_T_1 fcode x : map_T_1 fcode xs
```

Now, assume the following third application of `map`:

`map i2b xs` where `i2b :: Int -> Bool`

The previous apply function cannot be used because it does not match the type of `i2b`. We create the following additional data type `T_2` and an apply function for it (rules **RemoveHOTypes** and **Encode**):

```
data T_2 = C_i2b

apply_T_2 :: T_2 -> Int  -> Bool
apply_T_2 code x = case code of
                     C_i2b -> i2b x
```

Then, the specialized version of `map` has to be used and apply functions have to be inserted (rule `ApplVar`):

```
map_T_2 :: T_2 -> [Int] -> [Bool]
map_T_2 fcode []     = []
map_T_2 fcode (x:xs) = apply_T_2 fcode x : map_T_2 fcode xs
```

Finally, the application is replaced by:

```
map_T_2 C_i2b xs
```

4.7.4 Comments

Note that, after the HOE, all function applications are saturated with arguments, such that the result is not a function. Also, no argument to a function is a function. In principle, one could now replace all curried definitions and applications by tuple representations. This is not done in the $\mathcal{HDC}$ compiler for two main reasons:

1. The tuples, which are objects of the $\mathcal{HDC}$ language, are, in turn, expressed in terms of pattern matching `case` expressions, which require curried functions on the right hand side again.

2. The interpreter can remain simpler if it only has to deal with curried functions.

We adopt the following convention: after the HOE, any application of an $\mathcal{HDC}$ function has to supply all curried arguments. This schema can be regarded as first-order and is isomorphic to a schema of tupled arguments.

$$
\begin{array}{l}
f_i :: t_{(i,1)} \texttt{ -> } t_{(i,2)} \texttt{ -> } \ldots \texttt{ -> } t_{(i,m(i))} \texttt{ -> } t_{(i,0)} \\
f_i \; arg_{(i,1)} \; arg_{(i,2)} \; \ldots \; arg_{(i,m(i))} \texttt{ = } body_i
\end{array}
$$

can be emulated by a new function

$$
\begin{array}{l}
f' :: \texttt{Data -> Data} \\
f' \; arg \\
\quad \texttt{= case } arg \texttt{ of} \\
\qquad\qquad C_1 \; arg_{(1,1)} \; \ldots \; arg_{(1,m(1))} \texttt{ -> } CR_{t(1,0)} \; body_1 \\
\qquad\qquad \vdots \\
\qquad\qquad C_n \; arg_{(n,1)} \; \ldots \; arg_{(n,m(n))} \texttt{ -> } CR_{t(n,0)} \; body_n
\end{array}
$$

Figure 7: Elimination of mutual recursion by emulation

4.8 Elimination of mutual recursion

The $\mathcal{HDC}$ compiler implements two methods for the removal of mutual recursion in programs: *elimination by inlining* and *elimination by emulation.* Mutual recursion is identified by calculating the strongly connected components (SCCs) in the graph of functional dependences. Since there is no mutual recursion between SCCs, the methods can be applied to each SCC independently.

4.8.1 Elimination by inlining

This method can only be used for an SCC which contains a node f whose removal from the SCC would make the residual graph s acyclic. The set of functions which s represents is, therefore, free of mutual recursion. Thus, it is possible to inline all calls of functions in s in the body of f, until the only recursive calls left are directly recursive (Kaser et al., 1993).

4.8.2 Elimination by emulation

If all mutual recursion in a program is to be removed, an alternative approach has to be taken for SCCs in which mutual recursion cannot be eliminated by inlining. It is always possible to transform an SCC to a supernode. The function associated with a supernode emulates the work of all functions of the SCC by encoding the actual parameters and decoding the formal parameters. Let f_i, $1 \leq i \leq n$, be the functions of an SCC and $m(i)$ the number of arguments of function f_i. Function f', which emulates the f_i, is given in Fig. 7.

To avoid type conflicts, it is necessary to create, for each function f_i of the SCC, a constructor C_i to encode the arguments and a constructor $CR_{t(i,0)}$ for the result. The constructor name is used to select the body of function f_i. Finally, calls to the functions f_i have to be adapted to fit f'.

Whenever possible, elimination by inlining should take precedence over elimination by emulation. Inlining does not spoil the structure of the program and the resulting intermediate code can usually be optimized more effectively.

Both methods are expensive if the program contains cycles of mutual recursion with more than three to four functions. Unfortunately, cycles may be introduced by the transformations of earlier compilation phases. If programs are getting too big, due to the removal of mutual recursion, the elimination process can be turned off by setting a compiler switch. The default is to apply elimination by inlining, where possible, and then use the alternative method for the remaining mutual recursions. The $\mathcal{HDC}$ programmer should prefer the use of skeletons to user-defined recursive functions in order to keep the amount of recursion low.

4.9 `case` elimination

Pattern matching is not available in lower-level programming languages, such as C or Assembler, which are suitable for the target code of $\mathcal{HDC}$. Providing a run-time system for pattern matching would cause too much overhead. Therefore, we eliminate `case` expressions. We replace a `case` expression by nested `if` expressions. The branches of the `if` expressions contain the former right-hand sides of the `case` branches.

The variables introduced by patterns are replaced by a special expression, `ENth`, which is used to access a specific parameter of a constructor. `ENth` i $(C\ x_1\ \ldots\ \ x_n)$ returns x_i, $1 \leq i \leq n$. `ENth` 0, applied to a constructor, returns the constructor index in the function declarations. Note that `ENth` cannot have a valid Hindley-Milner type and, therefore, cannot be an $\mathcal{HDC}$ function! It is only used internally as a representation for an infinite set of type-correct functions. A former `case` branch is selected by an `if` expression, if the index of the constructor used in the pattern and the constructor index of the selector *sel* (Sect. 2.4.12) are the same.

`case` expressions which have only one branch receive a special treatment: no `if` expression is needed, assuming that the branch will always match.

4.10 Generation of intermediate DAG code

The syntax tree of each function is transformed to a directed acyclic graph (DAG) to enable sharing of common subexpressions. A DAG contains a set of expressions with associated numbers, ordered by their dependences. Subexpressions are referenced by the corresponding numbers. Fig. 8 shows an example DAG. The direction of the references is inverse to that of the data flow, which is depicted by the arrows in the figure.

The transformation of a syntax tree into a DAG is by a standard technique called the *value number method* (Aho et al., 1986). The nodes are enumerated such that the source of each data dependence has a smaller number than the target. The transformation proceeds by a bottom-up traversal of the syntax tree. The subject of the transformation of each node is an expression, in which subexpressions have already been converted to numbers by recursive application of the algorithm. It returns a number for the node as follows: if there is already an expression in the DAG that matches the input, then the number of the

```
sqrplus ::  [Int] -> Int
sqrplus xs = (xs!!0)*(xs!!0) + (xs!!1)*(xs!!1)

        0:  xs                  :: [Int]
        1:  0                   :: Int
        2:  (!!)    ↑0  ↑1      :: Int
        3:  (*)     ↑2  ↑2      :: Int
        4:  1                   :: Int
        5:  (!!)    ↑0  ↑4      :: Int
        6:  (*)     ↑5  ↑5      :: Int
        7:  (+)     ↑3  ↑6      :: Int
```

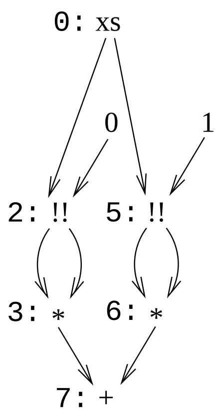

Figure 8: Example function definition with its DAG

existing expression is returned; otherwise, a new node for the expression is created and its number is returned.

Optimizing transformations (as described below) are performed at this intermediate code level.

4.11 Tuple elimination

Tuples are replaced by algebraic data types, one for each occurring tuple type. Each tuple is tagged with the appropriate constructor for its particular type. This simplifies the run-time system and, at the same time, provides fast access to information about the types and sizes of the components of the tuple by looking them up in a table, which is accumulated in file `funtable.c`. As a consequence, the memory management functions need not be specialized for each particular type.

4.12 Optimization cycle

Code optimization is done in a cycle. Each iteration performs three steps in sequence: inlining of functions calls, rule-based DAG optimizations and size inference.

The process of replacing the call of a function by its body, after substituting the actual for the formal parameters, is called *inlining*. We use inlining to enable further optimizations on DAGs, e.g., deletion of dead code and sharing of common subexpressions. Inlining is performed on DAGs in which common subexpressions appear only once. Due to the sharing of common subexpressions, there is no risk of duplicating work.

Inlining triggers common subexpression elimination for two reasons. First, it aggregates common subexpressions which have been spread across the program – maybe due to transformations made before, maybe due to the program itself. Second, it specializes variables by replacing the formal parameters of the function inlined by the actual parameters. This permits partial evaluation and checking for value equality rather than name equality when identifying common subexpressions. Value equality is a coarser equivalence relation, i.e., it induces more commonality.

The information gathered in the size inference is useful for the inlining heuristics of the following iteration of the optimization cycle and for the space-time mapping. Size inference has to be reapplied in each iteration because of the changes in the program due to inlining.

During each iteration, every DAG needed is processed as follows.

4.12.1 Inline expansion

The inline expansion transforms a source DAG into a corresponding target DAG with possibly inlined calls. First, the nodes of the source DAG are copied successively. If a node representing a function call is reached, a heuristic decision, based on the expected amount of code increase, is made as to whether to inline this call or just copy the call node. In the case of inlining, the copying process switches its source temporarily from the caller to the callee. All nodes of the DAG of the callee will be copied. There is no recursive inlining of calls. Copied calls may be inlined in the next pass. Every time the inline expansion of a DAG for some function is completed, the body in the function definition is changed to the target DAG.

After inlining in the current pass is finished, the DAGs are simplified. Unused function arguments, except from apply functions which are called from skeletons, are deleted and dead code is removed. If it was possible to inline at least one call in the current pass and a specified maximum number of passes is not yet reached, inlining is repeated in the next pass. We chose two major strategies for inlining: *current version inlining* and *original version inlining*.

- **Current version inlining**
  The most recent DAG for the called function is inlined. This method requires fewer inlining operations, since DAGs with already inlined calls are used for inlining again. One drawback is that the functions are growing very fast and, therefore, the inlining process may be suppressed after only a few steps.

- **Original version inlining**
  The original definition of the function is inlined. This incurs a linear code growth when inlining recursive functions. Original version inlining offers more possibilities for optimization and, therefore, may lead to better results (Kaser et al., 1992).

Kaser et al. (1992) also compare static and profile-based approaches. At present, we do not accumulate or exploit profiling information.

Common subexpressions are eliminated during the conversion of the syntax tree into a DAG, which has already been described in Sect. 4.10. However, new common subexpressions may appear during the inlining of a call. Since new nodes introduced by the inlining process are always created with the same function as used for DAG construction, no unshared common subexpressions will be created.

$$
\begin{array}{rcl}
\textbf{intr-sinGen} & & \texttt{map}\ f\ xs \\
\{xs :: \texttt{[Int]}, xs \text{ regular}, \underline{\text{fresh}}\ i\} & \overset{\rightarrow}{=} & \texttt{sinGen (\\}i\ \texttt{->}\ f\ \texttt{(}xs\texttt{!!}i\texttt{)) (length}\ xs\texttt{)} \\
\\
\textbf{elim-sinGen} & & \texttt{(sinGen}\ f\ n\texttt{) !!}\ i \\
 & \overset{\rightarrow}{=} & f\ i
\end{array}
$$

Figure 9: Optimization rules for `sinGen`

4.12.2 Rule-based DAG optimizations

In this step, various algebraic optimizations can be applied. In the interest of brevity, let us focus here on optimizations in the context of space-time mapping (Sect. 4.14).

In numerical algorithms, in a call `map` f xs, the list xs is often of type `[Int]` which defines a set of indices. In the simplest case, it is an arithmetic sequence (`[`a`..`b`]`) or obtained from index set transforming functions. Since enumerations of index sets are, in general, too inefficient, we represent them by functions (this requires a certain regularity). We can generate an index set from its describing function by a skeleton named `sinGen`, see Sect. 3.3.2. It takes a function f, which describes an index set, and an integer n and delivers a list of length n, in which the ith element is defined by applying f to i. Fig. 9 contains two optimization rules: one which introduces and one which eliminates `sinGen`. Note that there is some potential for fusion (similar to `map` fusion), e.g.:

$$
\begin{array}{rcl}
 & & \texttt{map}\ f\ \texttt{(sinGen}\ g\ n\texttt{)} \\
\{\text{intr-sinGen}, \underline{\text{fresh}}\ i\} & \overset{\rightarrow}{=} & \texttt{sinGen (\\}i\ \texttt{->}\ f\ \texttt{(sinGen}\ g\ n\ \texttt{!!}\ i\texttt{))}\ n \\
\{\text{elim-sinGen}\} & \overset{\rightarrow}{=} & \texttt{sinGen (\\}i\ \texttt{->}\ f\ \texttt{(}g\ i\texttt{))}\ n
\end{array}
$$

These optimizations have the problem that higher-order arguments are reintroduced (e.g., for the lambda expressions introduced above). Of course, one could apply such optimizations before HOE but, at this point, they would miss the applications that are enabled by the inlining specializations coming later.

4.12.3 Size inference

The size inference algorithm derives symbolic information about the result returned by a function from the values of structural variables which represent the symbolic information of its arguments. The goal is to improve the decisions made during each iteration of the optimization cycle and to determine automatically a space-time mapping at compile time, if possible.

We are interested in the following symbolic information about an $\mathcal{HDC}$ function:

1. the size of the result – in the case of nested lists a comprehensive description of all levels (Herrmann and Lengauer, 1998),

2. the number of operations,

3. the length of the longest path in the DAG, if all calls are expanded,

4. the number of steps for a given number of processors, if the communication cost is disregarded – this can be estimated from the number of operations and the path length, using Brent's theorem (Quinn, 1994).

The size inference computes an abstract version of the $\mathcal{HDC}$ function, which takes the same number of arguments and has the same structure as the original function, but the operations it performs are abstract counterparts of the original operations. E.g., an abstract size operation for the append operator of plain lists is, simply put, "addition", because the size of the result is the sum of the sizes of both operands. The abstract version of a function application is the application of an abstract function to an abstract size.

The sizes are represented in symbolic form, as objects of an algebraic data type `Size`, containing, e.g., the following constructors:

- `Con ::  Int -> Size` defines a constant size.
- `Var ::  String -> Size` defines a free variable.
- `Add ::  Size -> Size -> Size` adds two sizes.

If, e.g., `Con 2` is the size of the list `[3,4]` and `Var "x"` is the size of a list `x`, then the size of the list `[3,4] ++ x` is `Add (Con 2) (Var "x")`.

Abstract functions take variables representing symbolic expressions. E.g., the constructor `Add` above is an abstract function. Abstract functions can be composed of other abstract functions. E.g., the number of operations needed for a list append depends on the length of both argument lists. (That is the $\mathcal{HDC}$ append; for the usual sequential append, the length of the second list is immaterial.) From this point of view, underlying memory optimization techniques like sharing in DAGs, which are not visible at the level of intermediate code, are not considered. The structures are treated as if they were flattened and the abstract values obtained are upper bounds and not exact.

The abstract functions are expressed in terms of the abstract values of their arguments, in order to make size inference a local computation, independent of its context, and allow for a largely polymorphic implementation of the skeletons. If the amount of space in terms of memory cells or the amount of time in terms of clock cycles is of interest, the abstract function must be supplied with according context information.

Because of the complexity of the symbolic expressions involved, we believe that the size inference of recursive functions is beyond the capability of present-day mathematical tools.

It is assumed that all recursion is captured in skeletons which are supplied with the four types of size information stated above.

Size inference is applicable only if the structure analyzed does not depend on run-time data, e.g., if the length of lists does not depend on input values.

We see a use for a complete size inference mainly in functional programs which represent a static system, e.g., a hardware description.

4.13 Abstract code generation

The defining expression of an intermediate function is mapped to a DAG in order to facilitate the sharing of common subexpressions. As described in Sect. 4.10, each DAG is represented by a table: each node of the DAG corresponds to a table entry, and each directed edge of the DAG is represented at the entry of the source of the edge (the target of the data dependence) by the table index of the target node of the edge (the source of the data dependence).

The phase of abstract code generation switches the interpretation of a DAG: before, it is interpreted with a denotational semantics, afterwards with an operational semantics. The structure of the DAG also changes slightly: one type of node is eliminated and three other types are introduced.

Let us reflect on the denotational interpretation. Here, a DAG is interpreted by starting evaluation of a distinguished node, the *root*. The result of the root node is considered the result of the function represented by the DAG. If the evaluation of a node requires the result of another node, this node is visited and evaluated. There is a special kind of node for accessing formal arguments. `if` nodes require a special treatment: the value of the condition has to be tested, and then only one of both branches is evaluated.

In the operational interpretation, the evaluation proceeds by traversing the table entries in sequence. If the result of another node is required, it has already been computed and can be looked up in a previous table entry. The root node is the last entry in the table and contains the result of the function. The problem with the `if` nodes is that when they are reached (if ever!) both branches already have been evaluated, also the wrong branch. Therefore, a mechanism is implemented to skip nodes belonging to the wrong branch. If a DAG does not contain `if` expressions, it is used as abstract code without modification.

To accomplish the skipping of nodes, `if` nodes are eliminated and the following *control* nodes are introduced:

1. `BranchFalse` *cond* i tests the boolean value at node *cond* and continues the execution of the DAG from node i if the condition test yields *False*. Otherwise it has no effect.

2. `Jump` i continues the execution from node i.

3. `Selection` *cond* a b tests the boolean value at node *cond* and returns the value of node a if the test yields *True* and the value of b otherwise.

The abstract code for an `if` expression has the structure shown in Fig. 10. The pointer to the condition refers to a node above. The forward references of `BranchFalse` and `Jump` are filled with table indices, once they are known, i.e., when the recursive generation of abstract code for nested branches has finished. Jumps are introduced to skip over code in branches which are not reached due to the invalidity of their condition.

For reasons of soundness and efficiency, the conversion of a DAG to abstract code involves the duplication of some expressions that are shared in the intermediate DAG code – more precisely, of common expressions that are located in a set of nested `if` expressions

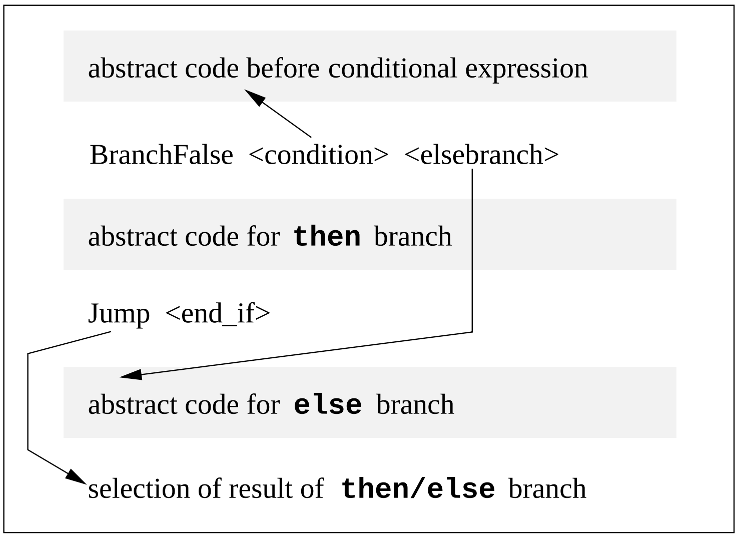

Figure 10: Abstract code structure for an `if` expression

in several, but not in all branches. These expressions are evaluated at most once and should not be shared.

The structure of nested `if` expressions can be maintained by an auxiliary tree T, whose nodes represent regions of the DAG. The root of T represents the entire DAG. Each occurrence of a conditional in the region, which a node represents, leads to two subtrees, one for the `then` branch and one for the `else` branch. All other nodes in the region (which do not belong to an `if` expression) are represented by a list of single pointers. Each node in T has an associated list with the pointers belonging to its subtree. Pointers to formal parameters are placed at the top level, so that they can be shared in all branches.

This arrangement has a high potential for duplicate pointers. To reduce duplication, the lists in each node are optimized by applying the following rules:

1. **Horizontal common expression elimination.** Pointers to expressions common to both branches of an `if` expression are moved up one level to the enclosing branch:

$$\begin{aligned} common &= pointers\_then\_branch \cap pointers\_else\_branch \\ pointers\_then\_branch' &= pointers\_then\_branch \setminus common \\ pointers\_else\_branch' &= pointers\_else\_branch \setminus common \\ pointers\_enclosing\_branch' &= pointers\_enclosing\_branch \cup common \end{aligned}$$

   This transformation is performed bottom-up, starting with the innermost `if` expressions. Therefore, a common pointer can be moved up further if the branch opposed to the enclosing branch contains the same pointer.

2. **Vertical common expression elimination.** Pointers to expressions already in the scope of an enclosing branch are removed in each list:

$$\mathit{pointers}' = \mathit{pointers} \setminus \mathit{already\_defined\_pointers}$$

already_defined_pointers is computed from the lists of all nodes on the path from the root to the current node in the tree of `if` expressions.

3. **Sorting**. For each node, the pointers (except pointers to `if` expressions) are sorted by their dependences. In the sorted list, the pointers to `if` expressions are placed at the earliest point at which all required pointers have been defined.

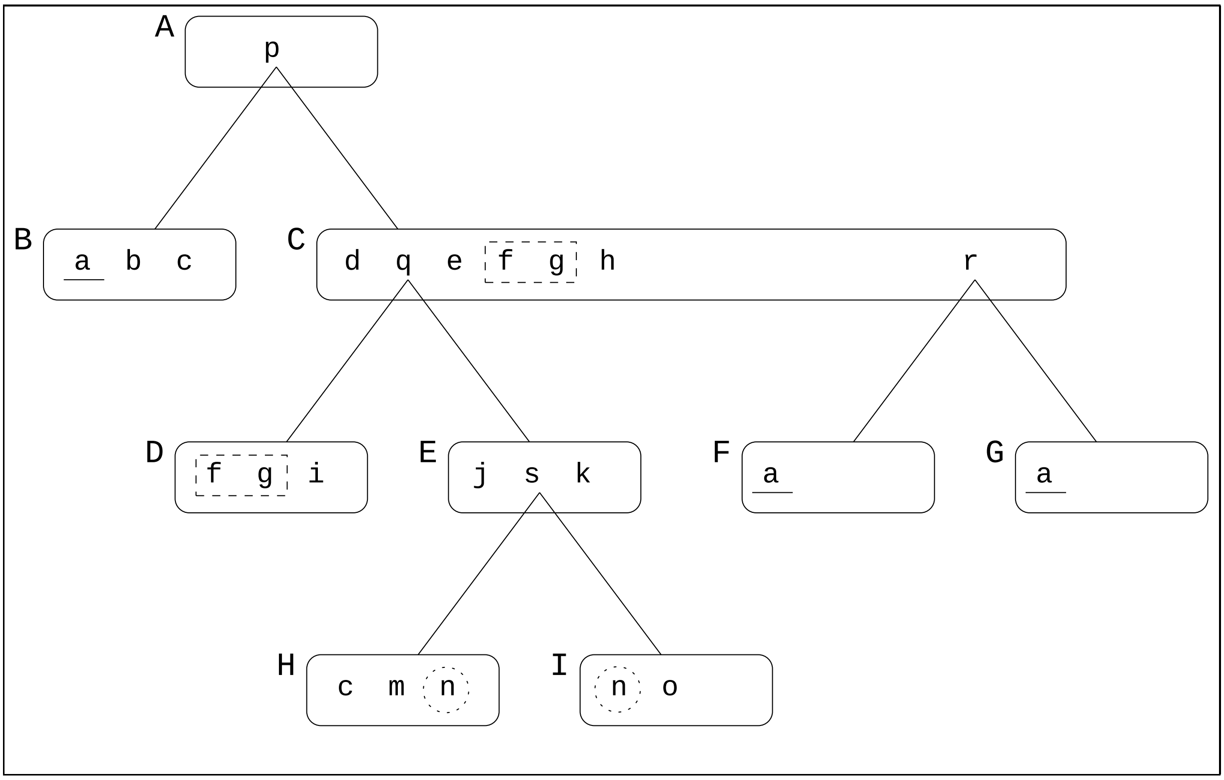

Figure 11: Example tree of `if` expressions

The rules are applied in the order stated because horizontal elimination possibly enables additional vertical elimination, but not vice versa. We sort at the end, since the other optimizations may change the lists and could violate the ordering. Note that the optimizations do not analyze `if` conditions.

Example:

Fig. 11 shows a tree of `if` expressions. The nodes A,...,I contain lists of pointers to expressions used in the parts of the corresponding DAG, named with lowercase letters. p,...,s are pointers to `if` expressions. The horizontal common subexpression elimination will move n from nodes H and I to node E. a will be moved from F and G to C. a is now common to both branches of A, thus it will be moved from B and C to A. The vertical common subexpression elimination will remove f and g from node D. In the abstract code of D, the pointers f and g from C are used to evaluate the expressions. To facilitate this, the sort must place f and g before q in the list of C. In order to reuse the pointer of c, it would have to be defined in node A. Without analyzing the conditions, it is not clear whether the evaluation of the corresponding expression is necessary. Thus, the expression will be duplicated in the abstract code, i.e., contained in B and H.

Using the information of the lists (which expressions must be defined in what branch) and the DAG code (list containing the expressions), the abstract code can be generated recursively for all nested `if` expressions.

4.14 Space-time mapping

A space-time mapping is a one-to-one mapping from a domain of computation points to the cartesian product of discrete space and time. The space part is known as *allocation*, the time part as *schedule*. The technique of space-time mapping has a long tradition in loop parallelization (Lengauer, 1993). Some of the ideas can be adapted to `while` loops (Griebl and Lengauer, 1994; Collard, 1995) and even to non-linear recursion (Herrmann and Lengauer, 1996). However, the structure of dependences we are encountering makes integer linear optimization, which is the central search method for a space-time mapping in loop parallelization and which has the nice property of yielding the best solution in the considered search space, unsuitable for general $\mathcal{HDC}$ programs.

Space-time mapping is most effective when applied in the individual derivation of parallel skeleton implementations. This approach is described in detail in Herrmann and Lengauer (1996). If the dependence structure of the skeleton is sufficiently regular –as, e.g., for some kinds of $\mathcal{DC}$– the points of computation can be laid out in time and space at compile time. The size of the computation space will depend on the problem size and the number of processors, but its shape will not (Herrmann and Lengauer, 1996).

The user is well advised to construct his/her program by composition of appropriate skeletons, which have been space-time mapped efficiently. Note that the generation of each skeleton is done by a Haskell function which is to be delivered by the skeleton implementer. It is up to this Haskell function, to use the results of the size inference provided or even to call external tools. The task of the $\mathcal{HDC}$ compiler is, at a minimum, to transmit symbolic space-time mapping information via the call structure of the program to the points where it is needed, by making use of the abstract functions delivered by the skeleton implementer. The nodes of each DAG are scheduled sequentially by the compiler, no parallelization is done in this phase.

4.15 Code generation

The code generation phase first produces C code, which is then compiled with a standard C compiler and linked together with the functions of our run-time library, which are also written in C. The C code is generated in two phases. First, the abstract code of the user program is translated; see Sect. 4.15.1. Second, an appropriate implementation is generated for each skeleton instance used in the program; see Sect. 4.15.2.

4.15.1 DAG compilation

For each DAG of the abstract code, a C function is generated and appended to file `code.c`. Each node in the DAG is treated seperately. Each constructor used is inserted into a table, in file `funtable.c`, to provide the run-time environment with the necessary information about types and sizes of the components of the object it constructs. For each call of a skeleton, the name of the skeleton together with the actual types of the arguments are stored.

4.15.2 Skeleton generation

After all DAGs have been processed, the instantiations of the skeletons are generated and stored in file `skel.c`.

$\mathcal{HDC}$ offers a special, very flexible mechanism for the integration of custom-implemented skeletons. For each skeleton, the implementer delivers a Haskell function, say, Φ, which is called by the code generator of the $\mathcal{HDC}$ compiler and which produces the actual instance of the skeleton. In the simplest case, the body of function Φ will be just a Haskell string of C target code, but Φ can also prescribe decisions based on type and size information provided by the compiler.

Remember that the C code generated must be monomorphic; this applies also to the implementation of a skeleton. Thus, the programmer of Φ has to consider at least the root symbol of the type tree of each argument, i.e., the implementation must differ, e.g., between lists and integers, but not necessarily between lists of `Int` and lists of lists of `Int` since, in the latter case, the root of the type tree is in both cases the list type constructor.

To illustrate what a generic skeleton implementation may look like, let us discuss an abstract version of the implementation of the `map` skeleton in a parallel model, in which all data is passed along with the control. The `map` skeleton takes a function (really a closure, i.e, the code of a function together with an environment) and a list and applies the function elementwise to the list.

For simplicity, we assume a space-time mapping which allots roughly the same number of list elements to each processor. This mapping is efficient if the amount of work is nearly equal for each element of the list.

One might consider the use of collective MPI operations (Pacheco, 1997) like *broadcast* (to distribute the function closure), *scatter* (to distribute the list among the processors) and *gather* (to collect the results from the processors). This would work for lists of `Int`, `Bool` or `Double` but would require special skeleton implementations for these types. In general, *gather* and *scatter* cannot be used, since they assume that lists are plain and do not contain references to a heap. E.g., if the elements are functions, the list contains just pointers to a shared heap. As a consequence, we have to custom-implement collective operations for $\mathcal{HDC}$, using the MPI primitives *send* and *receive*. Here, again, $\mathcal{DC}$ proves to be a useful technique.

4.15.3 Run-time library

The run-time library, which is comparatively small, contains the implementation of all functions which do not depend on the user program – especially, predefined functions which cannot be coded with a few C statements (those are inserted directly in the code), which perform memory management, and which pack and unpack data structures.

5 The Parallel Run-Time Environment

5.1 The model of parallel execution in $\mathcal{HDC}$

Our aim has been to provide a platform which does not limit the design choices concerning parallelism. Still, we are staying away from unstructured fork-join parallelism (Almasi and Gottlieb, 1989) and, where possible, make use of the $\mathcal{DC}$ paradigm. In the interest of generality and scalability, our execution model is SPMD.

The control structure is organized as follows. At the beginning, all processors form a single block. In a parallel computation, this block –let us called it the superblock– is divided into a number of subblocks and the *master* of the superblock sends a task to the masters of the subblocks. When the task a subblock is assigned to is terminating, its master sends an according signal to the master of the superblock. No tasks are initiated and no completion messages are sent across a superblock's border. We call this the principle of *communication-closed blocks*, in analogy to the principle of *communication-closed layers* (Elrad and Francez, 1982).

Another important issue is the data layout, i.e., the way in which the data is distributed among the processors. We distinguish three styles of data layout. The first applies to atomic data and to tuples. Only lists and algebraic data types, which can become large, are subject to the other distributions.

- **Centralized data layout:** All input data of a task is passed along with the signal of initiation of the task and all output data is passed back with the report on the task's completion. Obviously, this is a good choice if the amount of data is small, although it might incur some unnecessary communication. For large data, a centralized data layout will lead to unacceptable overhead, due to data transmission, or even to memory overflow.

In the remaining two layout styles, instead of passing the data with the control, only information about the location of the data is passed.

- **Hierarchical data layout:** The input and output data of each block is distributed among the processors assigned to the block, as prescribed by the space-time mapping. The default space-time mapping is that the data is distributed in balance across all available processors. This layout is especially convenient for $\mathcal{DC}$ algorithms on large data which does not fit onto a single processor, but only if the data size decreases with the division of the problem, as in the `dc4io` skeleton.

- **Globally distributed data layout:** The input data is distributed according to a space-time mapping. Each intermediate and result value is located on the processor that produces it. This is known as the *owner-computes rule* (Wolfe, 1995). To get data from another processor, a remote memory access (RMA) has to be performed. RMA involves a communication – however, it is not explicit and, thus, does not violate the principle of communication-closed blocks.

5.2 Organization

The available implementations of a skeleton determine the set of possible space-time mappings which can be chosen in a parallel execution. Thus, it is important to realize that skeleton implementations are generated in dependence of the context in which they are called, exploiting type and possibly also symbolic size information (see Sect. 4.15).

The first argument of each skeleton implementation, like the first argument of the other functions generated, contains a pointer to system information, comprising a description of the master and the current part of the topology the processor belongs to.

The user has the option to provide all functions with an additional explicit parameter, which contains a mapping strategy introduced in the source code. Skeleton implementations can use this strategy in the space-time mapping.

5.3 Interaction with skeleton implementations

If the computation is divided into subcomputations according to the $\mathcal{DC}$ paradigm, the block of processors is divided into subblocks. Each processor belongs to exactly one subblock. Each subproblem is solved on its own subblock. At the beginning, the block has one master processor, the other processors are *slave*s. The division of a block involves the creation of new masters by the old master, one for each subblock.

Let us now revisit the implementation of the `map` skeleton from Sect. 4.15.2. If the list does not contain at least two elements or the block has only one processor, `map` must be computed sequentially. (`map` *may* be computed sequentially if parallelization does not pay off according to a strategy chosen by the skeleton implementer.) Otherwise the block is divided into two subblocks; let us call them the *left* subblock and the *right* subblock. The left subblock computes `map` on the left part, and the right subblock on the right part. The processor responsible for the whole block before, say, L retains the responsibility for the left subblock and sends the packed function closure and the right part of the list to another distinguished processor, say, R responsible for the right subblock. Computation proceeds recursively until the left and right subblock are united again and R gives back control for its part to L.

Now, let us have a closer look at the call mechanism. Processor L is the one on which the `map` skeleton is called. Thus, it receives all formal arguments via a function call. Processor R is activated by L with an index of the actual skeleton instance. R uses this index to call a *slave skeleton*. This skeleton does not receive the application data via a function call, because this data is not yet available on R. Instead, the following interaction takes place: L sends the data to R, both L and R call the master skeleton with their particular subproblem, R returns into the slave skeleton and sends its result back to L. Note that Φ of Sect. 4.15.2 has to generate two skeleton instances here: the one for the master and the one for the slave!

6 Examples

6.1 Karatsuba's polynomial product

This subsection contains material, which we have published before with respect to a slightly modified $\mathcal{DC}$ skeleton (Herrmann and Lengauer, 1996).

In 1962, Karatsuba published a $\mathcal{DC}$ algorithm for the multiplication of large integers of bitsize N with cost $O(N^{\log_2 3})$ ($\log_2 3 \approx 1.58$) based on ternary $\mathcal{DC}$ (Aho et al., 1974). A trivial algorithm has complexity $O(N^2)$. As an example of ternary $\mathcal{DC}$, we choose the polynomial product, which is the part of Karatsuba's algorithm that is responsible for its complexity.

Here, we concentrate on the product of two polynomials which are represented by powerlists (Misra, 1994) of their coefficients in order. The length of both lists is the smallest power of 2, which is greater than the maximum of both degrees. We consider $+$, $-$ and $*$ operations on polynomials; when applying them to integers, we pretend to deal with the respective constant polynomial. If a, b, c and d are polynomials in the variable X of degree at most $N < 2^{n-1}$, then $(a * X^N + b) * (c * X^N + d) = h * X^{2*N} + m * X^N + l$, where $h = a * c$ (h is for "high"), $l = b * d$ (l is for "low") and $m = (a * d + b * c)$ (m is for "middle"). The ordinary polynomial product uses two polynomial subproducts to compute m, leading to quadratic cost, whereas the Karatsuba algorithm uses the equality $m = (a + b) * (c + d) - h - l$ to compute only a single additional polynomial subproduct. Polynomial addition and subtraction does not influence the asymptotic cost because it can be done in parallel in constant time and in sequence in linear time.

Due to the data type and data dependence restrictions imposed by our skeleton, the input vector of the skeleton is the zip of two coefficient vectors (`zip` $[a_0,\ldots,a_{2*N-1}]$ $[b_0,\ldots,b_{2*N-1}] = [(a_0, b_0),\ldots,(a_{2*N-1}, b_{2*N-1})]$) and the result is the zip of the higher and lower part of the resulting coefficient vector, as can be seen in the definition of `karatsuba`, which multiplies two polynomials represented by equal-size powerlists:

```
import HDCPrelude
infixl 2 ->>

-- polynomial product
-- the length must be a power of 2
-- using Karatsuba's algorithm

karatsuba :: [Int] -> [Int] -> [Int]
karatsuba a b =
  let basic (x,y) = (0,x*y)
      divide  [(xh,yh),(xl,yl)]
            = [(xh,yh),(xl,yl),(xh+xl,yh+yl)]
      combine [(hh,hl),(lh,ll),(mh,ml)]
            = [(hh,lh+ml-hl-ll),(hl+mh-hh-lh,ll)]
  in ilog2 (length a)                     ->> \n ->
     zip a b                              ->> \x ->
     dc4io 3 2 2 basic divide combine n x ->> \z ->
     map fst z ++ map snd z

(->>) :: a -> (a -> b) -> b
x ->> f = f x

ilog2 :: Int -> Int -- ceil of real log2
ilog2 n = if n<=1 then 0
                  else 1 + ilog2 ((n+1)`div`2)

parmain :: IO Unit
parmain = get >>= \a ->
          get >>= \b ->
          put ((karatsuba a b) :: [Int])
```

The operator `->>` forces a sequencing of computation steps. We use it to avoid multiple evaluations.

The first three arguments of `dc4io` are the degrees of the problem division, the input data division and the output data composition. Of the constituting functions, `basic` multiplies two constant polynomials. Function `divide` divides a problem into three subproblems: the first is working on the high parts, the second on the low parts and the third on the sum of the high and the low parts, corresponding to $(a+b)$ and $(c+d)$ of the formula for m. The function `combine` combines the results (`hh`,`hl`) (the high parts), (`lh`,`ll`) (the low parts) and (`mh`,`ml`) (the middle parts). The high positions `mh` of the middle parts overlap with the low positions `hl` of the high parts, and the low positions `ml` of the middle parts with the high positions `lh` of the low parts. Results of overlapping positions have to be summed. Further, the results of the high and low part have to be subtracted from the

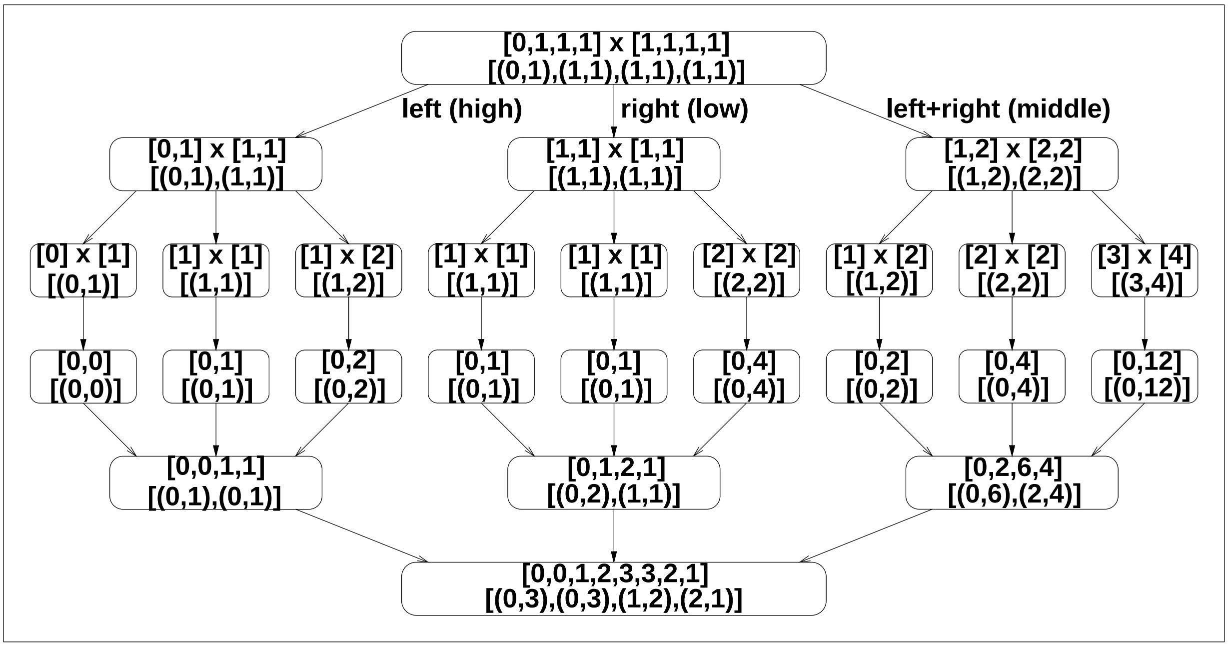

Figure 12: Call graph for a call of `karatsuba`

result of the middle part. As an example, Fig. 12 depicts the call graph for multiplication of the polynomials $(X^2 + X + 1)$ and $(X^3 + X^2 + X + 1)$ whose result is the polynomial $(X^5 + 2 * X^4 + 3 * X^3 + 3 * X^2 + 2 * X + 1)$. In each node, we give on top the polynomials as lists of their coefficients and below the representation as required by the skeleton.

6.2 Frequent set

The frequent set problem (Toivonen, 1996) belongs to the application area of *data mining*. Consider the following application example in a supermarket. Sets of articles, which are often purchased together, should be shelved close to each other. These sets should be obtained from statistics data compiled at the point of sale. Let M be the set of all articles, for simplicity enumerated from 0 to m. Let the database be a bag of subsets of M; each element of the bag contains a set of articles of a single bill. The number of occurrences of a single article is of no interest, but the number of occurrences of each subset of M is. The task is to report all subsets of M which are *frequent*, i.e., appear in (are a subset of) more than a certain fraction of all bag elements, called the *threshold*.

We present a straight-forward algorithm for the frequent set problem in $\mathcal{HDC}$, derived from Alg. 3.7 of Toivonen (1996).

A more sophisticated and efficient algorithm was derived by Hu (1999). We assume that subsets of integers are represented in a list in increasing order. The input consists of the threshold and the list of bills. The output is the list of frequent sets.

Our example contains the following functions:

- `compareSet` compares two sets with respect to a particular ordering on the subsets of integers (first by length, then by lexicographic order (Aho et al., 1974)). We use the `while` skeleton to terminate the comparison as soon as the result is known.
- `isElem` checks whether an element is in the set.
- `isSubSet` checks for the subset property.
- `insertSet` adds a new element to an ordered set.
- `remDuplicates` removes duplicates.
- `countSubsets` counts the number of occurrences as a subset in the bag.
- `fracOK` checks whether the fraction of bag elements the set appears in as a subset exceeds the given threshold value.
- `freqSets` constructs all frequent sets.
- `datamineSet` constructs all frequent sets of cardinality i, ordered by increasing i.
- `datamine` is the entire algorithm without input/output actions.

```
import HDCPrelude

compareSet :: Ord a => [a] -> [a] -> Int
-- compares two sets with respect to first the size, then the
-- lexicographic ordering, delivers -1 if xs<ys, 0 if xs=ys, 1 if xs>ys
-- the lists representing the (unordered) sets must be sorted
compareSet xs ys
 = if length xs == length ys
      then let firstdiff =
                skel_while (\i -> if (i<length xs) then (xs!!i==ys!!i)
                                                    else False ) (+1) 0
           in if firstdiff == length xs
                 then 0
                 else if xs!!firstdiff > ys!!firstdiff then 1 else (-1)
      else if length xs > length ys then 1 else (-1)

isElem :: Ord a => a -> [a] -> Bool
-- checks if e is element of the set s
isElem e s = any (==e) s

isSubSet :: Ord a => [a] -> [a] -> Bool
-- checks if sub is a subset of super
isSubSet sub super = all (\s -> isElem s super) sub
```

```
insertSet :: Ord a => a -> [a] -> [a]
-- adds x to the set xs
insertSet x xs = filter (<x) xs ++ (x : filter (>x) xs)

remDuplicates :: Ord a => [[a]] -> [[a]]
-- removes all duplicates in a set of sets
remDuplicates
 = let pivot xs = xs!!(length xs `div` 2)
       p xs = length xs < 2
       b xs = xs
       d xs = let less
                   = filter (\x -> compareSet x (pivot xs) == (-1)) xs
                  greater
                   = filter (\x -> compareSet x (pivot xs) == 1)    xs
              in [less,greater]
       c xs [as,bs] = as ++ (pivot xs : bs)
   in dc0 p b d c

countSubsets :: [Int] -> [[Int]] -> Int
-- counts the number of elements in bs which b is a subset of
countSubsets b bs = length (filter (isSubSet b) bs)

fracOK :: [[Int]] -> Double -> [Int] -> Bool
-- given a bag of sets bs, a fraction f and a set b
-- tells if b occurs as a subset in more than the fraction f of
--       all elements of bs
fracOK bs f b = ( fromInt (countSubsets b bs) / fromInt (length bs)) > f

freqSets :: [[Int]] -> Double -> [Int] ->
            ([[Int]],[[Int]]) -> Int -> ([[Int]],[[Int]])
-- given
-- bbag: the bag of sets
-- frac: the fraction of occurrences
-- sngl: all singleton frequent sets
-- (f,filast): the current/previous collection of frequent sets
-- i: the current level
freqSets bbag frac sngl (f,filast) i
 = (\fi -> (f++fi,fi))
   (filter (fracOK bbag frac)
           (remDuplicates
               (filter (\xs -> length xs == i)
                       [ insertSet s1 s2 | s1 <- sngl, s2 <- filast])))
```

```
datamineSet :: Int -> Int -> [[Int]] -> Double -> [[Int]]
-- data mining on the Set representation to enable
-- efficient parallel subset testing
datamineSet m u bbag frac
 =  (\siOK ->
     fst (foldl (freqSets bbag frac (map (!!0) siOK))
                (siOK,siOK) [2..u]))
    (map (\x->[x]) (filter (\x -> fracOK bbag frac [x]) [0..m]))
-- list of single items that satisfy fraction condition

datamine :: [[Int]] -> Double -> [[Int]]
-- the entire datamining algorithm
datamine bag frac =
  let maxi = red max 0 (concat bag)
      ubnd = red max 0 (map length bag)
  in  datamineSet maxi ubnd bag frac

parmain :: IO Unit
parmain = get >>= \threshold ->
          get >>= \bag ->
          put (datamine bag threshold)
```

7 Experimental Results

We have tested both examples with and without optimizations. In the tables in this section, the entry "no opt." refers to the following menu setting for the optimization phase of our compiler (Sect. 8.1.4).

```
Mutual Recursion Elim.: NONE
Optimize By Inlining:   False
```

For the entry "opt.", the menu settings were:

```
Mutual Recursion Elim.: ALL, by inlining if possible
Optimize By Inlining:   True
A -> Algorithm:         current version inlining
B -> Number of Loops:   4
C -> Max. Rel. Funsize: 4 * original size
D -> Max. Abs. Funsize: 200 nodes
E -> Inline Order:      InlOrderCallGraph
```

7.1 The effect of optimizations

In Tab. 3, we have recorded some static characteristics of the code (line by line, as the compilation proceeds) and the effect of optimization.

The source programs contain *global* functions, which are visible in the entire program, and *local* functions (also constants), which are defined using `let` (line 1). We do not count lambda abstractions, partial applications, etc.

Functions of the prelude are not counted as part of the source code, but are counted as functions before HOE (line 4) and onwards.

The HOE phase has two different opposing effects: (1) functions which are not used are deleted and (2) polymorphic functions which are used in different contexts are duplicated, and apply functions are introduced to decode functional arguments. Line 5 lists the total number of nodes in all syntax trees. DAG generation leads to a reduction of this number (line 7). This is due to common subexpression elimination, which is always done, even if no inlining is performed. The number of functions in the upper part of the table include the interface definitions of the skeletons used (lines 3 and 4) which are skipped in the set of DAGs (line 6).

	number of	Karatsuba		frequent set	
1.	source functions	4 global, 3 local		11 global, 10 local	
2.	source lines	30		86	
3.	functions before HOE	75		104	
4.	functions after HOE	37		103	
5.	tree nodes	416		968	
	number of	no opt.	opt.	no opt.	opt.
6.	DAG functions	31	11	86	25
7.	total DAG nodes	202	269	492	455
8.	total abscode nodes	212	343	534	563
9.	lines `code.c`	565	476	1492	850
10.	lines `code.s`	1313	1148	2972	2709
11.	lines `skel.c`	121		272	
12.	lines `skel.s`	695		1151	

Table 3: Effect of optimizations

Inlining also has two different effects. On the one hand, nodes are duplicated if inlined into more than one function. On the other hand, the function inlined may be deleted and optimizations apply, which have been enabled by the inlining. Line 8 shows the number of nodes after generation of abstract code, which has grown by the number of control nodes that have been inserted to make conditionals non-strict.

The code generated from the DAGs is in file `code.c`; `code.s` is the assembler file generated with the C compiler. The skeleton implementations are generated independently

problem	size	$\mathcal{HDC}$ compiler no opt.	$\mathcal{HDC}$ compiler opt.	**GHC**
Karatsuba	512×512	1.9	1.9	1.6
	1024×1024	5.9	5.9	4.8
frequent set	0.05/100	8.4	6.6	3.0
	0.05/200	15.6	12.0	4.5
	0.02/100	34.9	27.2	14.8
	0.02/200	63.2	50.0	20.8

Table 4: Sequential execution times

from the DAGs. Therefore, optimization has in general no effect on them – except for cases in which optimizations eliminate a call to a skeleton.

7.2 Sequential execution times

We measured the execution time of the examples Karatsuba and frequent set and compared them with **GHC** (V. 4.01), a compiler producing very fast lazy code. We compiled all our C sources with the **GNU** C compiler (V. 2.7.2.3) and optimization level –O3 (without/with optimization refers only to the $\mathcal{HDC}$ compiler), **GHC** was used with option –O.

We gave **GHC** a straight-forward program for the Karatsuba algorithm, which is not based on the special skeleton `dc4io`. It can be found in Appendix B. In the frequent set program, only the input/output functions were adapted to Haskell, and the skeletons `dc0` and `while` were supplied with their recursive definitions which do not cause as much overhead as the recursive definition of `dc4io`.

The size in the Karatsuba example indicates the length of both polynomials. For the frequent set example, we state the threshold value and the number of elements in the bag (the times still depend on the particular sample chosen).

All times in Tab. 4 are given in seconds of pure process time on a SUN workstation "Sun UltraSPARC" 167 MHz CPU with 256MB of memory.

We expect that these numbers can still be improved. E.g., currently, we have no destructive update mechanism for lists. Thus, each modification results in a copy of the entire list.

The elimination of higher-order functions causes an overhead due to decoding, which is especially notable if many higher-order functions are involved, as, e.g., in the frequent set example. Inlining can do a good job here, but some overhead will remain.

7.3 Potential for Parallelism

The parallel skeleton implementations and the run-time system have yet to be developed to a point where speedup experiments are possible. Thus, we can only provide an idea of the potential for parallelism with data extracted by our compiler and interpreter.

input	no. of operations			no. of par. steps			average par.		
samp/th.	no opt.	opt.	ratio	no opt.	opt.	ratio	no opt.	opt.	ratio
$A/0.5$	12075	8782	0.73	355	224	0.63	34.0	39.2	1.15
$B/0.5$	55935	39559	0.71	893	586	0.66	62.6	67.5	1.08
$B/0.2$	360963	252887	0.70	1854	1239	0.67	194.7	204.1	1.05

Table 5: Run-time characteristics of the frequent set example

The Karatsuba example is expressed with a skeleton whose parallelism is completely static, except for some parameters, e.g., the problem size. Thus, the potential for parallelism can be analyzed by hand, e.g., the maximum degree of parallelism for a polynomial product of size $2^n \times 2^n$ equals the number of base cases, which is 3^n.

The frequent set example is much more dynamic: optimizations can affect the structure of the entire implementation. Therefore, it pays to analyze the properties of the program after different phases of the compilation with the $\mathcal{HDC}$ interpreter. Tab. 5 shows the results of an interpretation of the abstract code with two different samples

$A =$`[[1,2,4,7],[5,6,7,8,9],[1,2,3,7],[1,3,5,8,9]]` and $B = A$`++[[1,2,3,4,5]]`

combined with thresholds of 0.5 and 0.2. The improvements after optimization demonstrate the important role inlining plays after the HOE.

The large amount of work is due partly to the nature of the problem and partly to the lack of sophistication of the source program. Regardless of that, note that the optimizations reduce the number of operations by up to 30% and that there is a high potential of parallelism.

8 Using the $\mathcal{HDC}$ Compiler and Interpreter

8.1 Menu structure

The menu structure reflects the state the $\mathcal{HDC}$ system is in and therefore the actual menu changes after certain phases of the compilation. Note that the reason for restrictions concerning the change from one menu to the other is the memory optimization. Data of previous compiler phases which is not necessary for proceeding is deleted. An exception is the optimization menu because a reload for trying optimization with new parameters would be unacceptable.

Options are chosen by typing the character printed ahead. In the following enumeration, we prefix the character with `*`, if the availability of the option depends on the (sub)state of the system.

8.1.1 Initial menu

The initial menu appears after starting the system. The following choices can be made:

- T `Test`. The purpose is to check if the compilation works on example `.hdc` programs with example inputs and produces the correct results. The names of the examples to be tested are listed in `test/testfiles`, the input/output pairs for an example, say *example*`.hdc` is contained in `test/`*example*`.test`.
- L `Load File`. The pathname of an example program to be compiled is asked for, starting from the directory in which the $\mathcal{HDC}$ system was started. If the name is prefixed with `=`, it starts with the directory `examples`.
- I `Interpreter`. The interpreter takes input data interactively from the user and evaluates the program with it. The purpose of the interpreter is to examine dynamic properties of the code at a certain phase of the compilation, i.e., collect statistical data for a particular input pattern, e.g., the number of reductions. This gives us an estimate of the overhead introduced by certain phases of the compiler.
- C `Code Generation`. Performs all compilation steps up to the final code without any interaction.
- H `HO Elimination`. The phases of monomorphization and higher-order elimination are performed. Because the internal representation of the program changed, a higher-order elimination menu appears (Sect. 8.1.2).
- S `Settings`. Enters the settings menu (Sect. 8.1.4).
- R `Restart`. Provides the same state as after starting the system, also reloads the prelude files.
- Q `Quit`. Leaves the system.

8.1.2 Higher-order elimination menu

This menu can be entered only directly after a higher-order elimination has been done. No optimization has been done yet. There are the following choices. To proceed in the compilation process, the option `O` (`Optimization`) has to be chosen.

- I `Interpreter`. As in the initial menu.
- O `Optimization`. Performs the optimization phase using the settings that can be changed in the settings menu. After this phase, the program is present in two representations: in the syntax tree form with some transformations (elimination of mutual recursion and `case` elimination) and in the DAG form derived from it which contains the most optimizations. After this phase, the optimization menu (Sect. 8.1.3) is entered.

`S` `Settings`.

`R` `Restart`.

`Q` `Quit`.

8.1.3 Optimization menu

The purpose is to make experiments with different optimization strategies without passing the time consuming higher-order elimination phase again and again. There are the following possibilities:

`I` `Interpreter`. As in the initial menu, it is the last syntax tree produced that is interpreted.

`D` `DAG Interpreter`. This interpreter works on the DAGs after the optimization phase. The effect of different inlining strategies can be observed on examples by recording the free schedule and degree of parallelism with respect to a limited number of processors.

`N` `New Optimization`. A new optimization is made based on the first DAG version generated.

`P` `Profiling Series`. To examine the behavior of many optimization strategies without a huge amount of tedious user interaction, this choice can be made. User-definable Haskell functions linked together with the $\mathcal{HDC}$ system are called, which control the profiling, deal with errors that may occur and summarize and format the result, e.g., as a LaTeX table.

`C` `Code Generation`. Calls the code generation on the optimizations yet made and with the target architecture specified in the settings.

`*E` `Execute`. This option can only be chosen after code has been generated. If selected, the user is asked for the input data, it is written into the input file, the compiled code is applied to this file, and after execution the output file is displayed on the screen. This is of course much faster than with interpretation.

`S` `Settings`.

`R` `Restart`.

`Q` `Quit`.

8.1.4 Settings menu

Selecting an option can either cause a step in a cyclic shift of alternatives or a prompt for input of some number or name. In the case of a cyclic shift, the current selection is displayed.

- P `Print Style`. Selects the format in which functions of the program are displayed, e.g., only by type.
- G `Generate Code for`. The target architecture resp. the execution model are to be defined here.
- I `Interpreter Statistics`. A switch for the collection of additional information about the computation, e.g., the free schedule, number of reductions, degree of parallelism, etc. This can cause the interpretation to take very long.
- T `Trace DAG Interpreter`. Selects the trace mode of the DAG interpreter.
- N `Number of Processors`. Defines the number of processors which are used for computing schedule information in the interpreters.
- *M `Mutual Recursion Elimination`. Toggles between different strategies of mutual recursion, which enable resp. give priority to elimination by inlining (Sect. 4.8.1) and elimination by emulation (Sect. 4.8.2). After the first DAG generation, this option disappears because the syntax trees have been deleted.
- O `Optimize by Inlining`. Toggles the inlining mode. If inlining is switched on, the points (`A`) to (`E`) appear by which details of the inlining strategy can be defined.
- *A `A -> Algorithm`. Toggles between current version inlining and original version inlining.
- *B `B -> Number of Loops`. Asks for the maximum number of iterations of the optimization cycle.
- *C `C -> Max. Rel. Funsize`. Asks for the maximum factor by which a function is allowed to increase in the number of nodes. If this limit is reached, the version of the function before starting the current inlining process is restored. The same holds for `D`.
- *D `D -> Max. Abs. Funsize`. Asks for the maximum number of nodes which a function is allowed to reach due to inlining.
- *E `E -> Inline Order`. Toggles between different orders of functions in an inlining phase.
- V `Verbose Mode`. Set if detailed information during the optimization should be displayed.

1 `Print Functions`. Prints the functions of the syntax tree on screen or into a file.

*2 `Print DAGs`. Prints the DAGs after the optimization.

*3 `Print DAGs (before optimizing dags)`. Prints the DAGs before optimization.

S `Save Options`. The currents settings are saved for the next session.

L `Load Options`. Saved settings can be loaded.

Q `Quit Settings`. Quit the settings menu and returns to the menu from which it was called.

8.2 Interpreter

The purpose of the interpreter is to collect statistical data of the program representation at a certain phase of the compilation with respect to particular input data. The interpretation is very slow and if only the output data is of interest, the user is advised better to generate sequential code and to execute it interactively.

There are two versions of the interpreter: one which works on the syntax tree (simply called `interpreter`), the other operates on the DAG structure (called `DAG interpreter`).

8.3 Directory structure

To work with the $\mathcal{HDC}$ compiler, the user has to set the shell environment variable `HDC_ROOT` which defines the path of the working directory. This directory has the following subdirectories:

- `doc`. Contains all documentation, e.g., this report.
- `examples`. Contains a set of source programs. Each $\mathcal{HDC}$ program has the file extension `.hdc`. There are also Haskell programs with file extension `.hs`, which serve for debugging and comparison purpose and differ only slightly from the `.HDC` program with the same name.
- `experiment`: Target programs of the $\mathcal{HDC}$ compiler are written into this directory. Also, input and output files of the target program are located here.
- `imports`: Contains the prelude files, see Sect. 8.4.
- `lib`: Contains the run-time libraries to be linked with the output of the $\mathcal{HDC}$ compilation.
- `profile`: This directory contains variants of the Haskell file `Experiment.hs`, which belong to the compiler. This file contains functions which control a particular experiment series and produce, e.g., LaTeX output of the results.

- `src`: This is the source directory. All its files are either necessary to build the $\mathcal{HDC}$ compiler (`.ly`, `.hs`, `.lhs`) or to generate the run-time library (`.h`, `.c`). Also, the `makefile` is located here.

- `test`: Used for verifying the current release of the $\mathcal{HDC}$ compiler. The file `testfiles` contains a collection of those examples, for which the $\mathcal{HDC}$ compiler should work correctly before a new version is committed. The files with extension `.test` contain pairs of input and output data against which the compiled program is to be checked.

8.4 The prelude parts

The prelude is divided into four parts, which can be found in directory `$HDC_ROOT/imports`. `Prelude.hdc`, `HDCPrelude.hdc` and `SkelPrelude.hdc` are loaded by the $\mathcal{HDC}$ compiler initially. `HDCPrelude.hs` is needed if $\mathcal{HDC}$ programs are to be used as Haskell programs. The definitions contained in these files form part of the $\mathcal{HDC}$ language and, thus, should not be changed by the user. They can be extended by additional definitions if new skeletons are to be added.

1. `Prelude.hdc` contains some predefined Haskell functions which can also be used in $\mathcal{HDC}$. This part of the prelude is listed in Appendix A.

2. `HDCPrelude.hdc` contains additional Haskell functions, which are of special interest to us. In particular, skeletons like `dc4io` are defined here. This prelude part is not listed because the type definition and explanation of the skeletons have already been presented in Sect. 3.

3. `SkelPrelude.hdc` contains interface (type) definitions of skeletons. The name of a skeleton must have the prefix `skel_`.

4. `HDCPrelude.hs` contains the Haskell definitions of the additional functions (in `HDCPrelude.hdc`).

9 Related Work

There have been many approaches to skeletal and functional programming. We concentrate here on those which have been most successful and/or have had significant influence on our work.

Two functional languages have been designed explicitly with parallelism in mind; both make use of parallel vector operations. The focus of the language *Sisal* (Skedzielewski, 1991) is on numerical computations, using loops on arrays. For some programs, its performance is superior to FORTRAN. Sisal is compiled to a data flow graph language. The idea of our intermediate DAG language stems from Sisal. In contrast to Sisal, the focus of the language *Nesl* (Blelloch, 1992) is on recursive programs using nested sequences. Nesl is compiled to an intermediate language, which uses parallel vector operations. Both Sisal and Nesl do not use skeletons and do not permit higher-order functions.

The language GpH (Trinder et al., 1998) is an extension of Haskell with a new primitive `par`, to be used together with the Haskell primitive `seq` to prescribe where values are supposed to be computed in sequence or in parallel. However, in contrast to $\mathcal{HDC}$, aside from a restriction of the evaluation order via `seq`, no schedule and allocation can be defined in GpH. Instead, parallel processes are distributed dynamically. GpH puts no restriction on the use of higher-order functions in Haskell. The user can define new skeletons, using evaluation strategies specified with `seq` and `par`.

There is another difference to $\mathcal{HDC}$: in order to preserve laziness, the input data for a process is only sent partially – if evaluation proceeds, further data must be requested. However, due to its treatment of higher-order functions, GpH is the language which is most similar to $\mathcal{HDC}$.

The idea to use a skeleton for $\mathcal{DC}$ was introduced by Cole (1989). The group of Darlington at Imperial College has published a collection of functional skeletons for parallel programming (Darlington et al., 1993).

P3L (Bacci et al., 1995) is an imperative language, which uses skeletons at the top level but does not support functions as run-time parameters of the skeleton. David Busvine and Tore Bratvold presented in their Ph.D. theses (Busvine, 1993; Bratvold, 1994) extensions of ML with skeletons, but their use of higher-order functions is very restricted.

The language Eden (Breitinger et al., 1997; Galán et al., 1996) facilitates the definition of skeletons on top of Concurrent Haskell. Eden imposes no restriction on higher-order functions. Eden differs from $\mathcal{HDC}$ in that skeletons have more restricted signatures and, therefore, cannot be used as generally; skeleton instances have to be wired together using channels.

10 State of the Implementation

At present, all compiler phases other than the optional phase of size inference are implemented. One critical challenge for the language $\mathcal{HDC}$ is effective load balancing. We plan to exploit the information supplied by the size inference in this regard.

The parallel implementations of all skeletons other than `map` have yet to be coded. Therefore, we have to defer the presentation of speedup results. Initially, we shall provide implementations in the model of centralized input/output.

Previous experimental work has demonstrated the potential for good speedups using $\mathcal{DC}$ skeletons (Musiol, 1996).

A Prelude.hdc

```
--                        Prelude.hdc

-- This file contains predefined HDC types and functions that are
-- already predefined in Haskell. "primitive" declarations indicate
-- that the function has a builtin implementation

-- **********************
-- **  basic functions  **
-- **********************

-- undefined constant
primitive undefined :: a

-- constant function
const :: a -> b -> a
const c x = c

-- identity function
id :: a -> a
id x = x

-- function composition
(.) :: (b->c) -> (a->b) -> (a->c)
f . g = \x -> f (g x)

-- strict function: always evaluates x to normal form and then
-- applies f to the result. The normal form of a partially applied
-- function is its closure represented as algebraic data type
strict :: (a->b) -> a -> b
strict f x = skel_strict f x

-- *********************
-- ** input and output **
-- *********************

-- the data type of IO actions
data IO a = IO a

-- constructs an empty IO action which returns x
return :: a -> (IO a)
return x = IO x
```

```
-- combines an IO action which returns x and an
-- IO action f which takes x as an argument to
-- a larger IO action
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>=) (IO x) f = skel_strict f x

-- ***************
-- **  boolean  **
-- ***************

-- built-in constants:
-- False :: Bool
-- True  :: Bool

-- logical negation
primitive not :: Bool -> Bool

-- logical and and or, strict in the
-- second argument if (&&) resp. (||)
-- not inlined, be careful!
(&&), (||) :: Bool -> Bool -> Bool
a && b = if a then b else False
a || b = if a then True else b


-- ******************************
-- **  comparison and ordering  **
-- ******************************

-- less, less_or_equal, greater, greater_or_equal, unequal, equal
primitive (<), (<=), (>), (>=), (/=), (==) :: Ord a => a -> a -> Bool

-- minimum and maximum
primitive min, max :: Ord a => a -> a -> a

-- *************************
-- **  general arithmetic  **
-- *************************

-- unary negation:
--     - :: Num a => a -> a
```

```
-- addition, subtraction, multiplication
primitive (+), (-), (*) :: Num a => a -> a -> a

-- sum, product
sum, product :: Num a => [a] -> a
sum xs     = skel_red (+) 0 xs
product xs = skel_red (*) 1 xs

-- power to an integer number
primitive (^) :: Num a => a -> Int -> a

-- ******************************
-- **  arithmetic conversions  **
-- ******************************

-- conversion from Int to Double
primitive fromInt :: Int -> Double

-- conversion from Double to Int
primitive floor :: Double -> Int
primitive ceiling :: Double -> Int

-- **************************
-- **  integer arithmetic  **
-- **************************

-- integer division and remainder
primitive div, mod :: Int -> Int -> Int

-- *********************************
-- **  floating point arithmetic  **
-- *********************************

-- floating point division
primitive (/) :: Double -> Double -> Double

-- square root
primitive sqrt :: Double -> Double

-- exponential and logarithm to base e
primitive exp, log :: Double -> Double
```

```
-- trigonometrics
primitive pi :: Double
primitive sin, cos, atan :: Double -> Double

-- ***********************
-- **  tuple selection  **
-- ***********************

-- first element of a pair
fst :: (a,b) -> a
fst (x,_) = x

-- second element of a pair
snd :: (a,b) -> b
snd (_,y) = y

-- **********************
-- **  list operators  **
-- **********************

-- built-in constructors
-- empty list: []  :: [a]
-- list cons:  (:) :: a -> [a] -> [a]

-- length of a list
primitive length :: [a] -> Int

-- test whether list is empty
primitive null :: [a] -> Bool

-- append two lists
primitive (++) :: [a] -> [a] -> [a]

-- append all sublists to a single list
concat :: [[a]] -> [a]
concat xs = skel_red (++) [] xs

-- filter all elements out of a list fulfilling a predicate
filter :: (a->Bool) -> [a] -> [a]
filter p xs = skel_filter p xs

-- list indexing
primitive (!!) :: [a] -> Int -> a
```

```
-- generate list of integer sequence with given bounds
primitive enumFromTo :: Int -> Int -> [Int]

-- take/drop the first _ elements of a list
primitive take, drop :: Int -> [a] -> [a]

-- the first element of a list
head :: [a] -> a
head xs = xs!!0

-- the list without the first element
tail :: [a] -> [a]
tail xs = drop 1 xs

-- apply a function to all elements of a list
map :: (a->b)->[a]->[b]
map f xs = skel_map f xs

-- construct a list of pairs from two lists
zip :: [a] -> [b] -> [(a,b)]
zip xs ys = sinGen (\i -> (xs!!i,ys!!i)) (min (length xs) (length ys))

-- apply a function to all pairs of elements of two lists
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith f xs ys = sinGen (\i -> f (xs!!i) (ys!!i))
                         (min (length xs) (length ys))

-- check whether a predicate holds for all/any element of a list
all, any :: (a -> Bool) -> [a] -> Bool
all p xs = (skel_while (\i -> if i<length xs then p (xs!!i) else False)
                       (+1) 0) ==length xs
any p xs = not (all (not . p) xs)

-- reduces the elements of a list given a binary operator and a neutral
-- element from the left resp. right
foldl :: (a->b->a) -> a -> [b] -> a
foldl f e xs = snd (skel_while (\(i,_) -> i<length xs)
                               (\(i,x)-> (i+1, f x (xs!!i))) (0,e))

foldr :: (a->b->b) -> b -> [a] -> b
foldr f e xs = snd (skel_while (\(i,_) -> i>=0)
                       (\(i,x) -> (i-1, f (xs!!i) x)) (length xs -1,e))
```

B Karatsuba in Haskell

```
left  xs = take (length xs `div` 2) xs
right xs = drop (length xs `div` 2) xs

karatsuba :: [Int] -> [Int] -> [Int]
karatsuba xs ys =
  if length xs == 1
     then [0,(xs!!0)*(ys!!0)]
     else let xhs = left xs
              xls = right xs
              yhs = left ys
              yls = right ys
              hs  = karatsuba xhs yhs
              ls  = karatsuba xls yls
              ms  = karatsuba (zipWith (+) xhs xls) (zipWith (+) yhs yls)
              mhls= zipWith3 (\m h l -> m-h-l) ms hs ls
              q0  = left hs
              q1  = zipWith (+) (right hs) (left mhls)
              q2  = zipWith (+) (right mhls) (left ls)
              q3  = right ls
          in  q0 ++ q1 ++ q2 ++ q3

main :: IO ()
main = do
        s <- readFile "input"
        let (a,rest):_ = reads s    :: [([Int],String)]
            (b,_):_    = reads rest :: [([Int],String)]
            c = karatsuba a b
        writeFile "output" (show c)
        return ()
```

Acknowledgements

This work has been funded by the DFG under project RecuR2 and by the DAAD under exchange projects with Britain and Sweden. We thank the Paderborn Center for Parallel Computing for access to their Parsytec GCel-1024.

Thanks to John O'Donnell for many fruitful discussions, in which he convinced us to base our transformational approach for $\mathcal{DC}$ on equational reasoning in Haskell. Kevin Hammond pointed out the problem of a possible loss of sharing due to communication. The idea of skeleton `sinGen` stems from the work by Björn Lisper (1996). We were made aware of the frequent set problem by David Skillicorn, who posed it as a benchmark problem for a recent Dagstuhl seminar on high-level parallel programming (no. 99171).

Thanks also to Holger Bischof, Peter Faber, Martin Grajcar and Henry Kehbel for their expert help concerning C, MPI and operating system issues.

The Data Display Debugger, developed by Zeller and Lütkehaus (1996), saved us much time in analyzing the automatically generated C code.

References

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Series in Computer Science and Information Processing. Addison-Wesley, 1974.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, 1986.

George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* Series in Computer Science and Engineering. Benjamin/Cummings, 1989.

Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. P^{3}L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. *ACM SIGPLAN Notices*, 32(8):25–37, 1997. *Proc. ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'97).*

Richard Bird. *Introduction to Functional Programming using Haskell.* Series in Computer Science. Prentice Hall Europe, 2nd edition, 1998.

Guy Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, Department of Computer Science, Carnegie-Mellon University, 1992.

George H. Botorog and Herbert Kuchen. Efficient parallel programming with algorithmic skeletons. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96, Vol. I*, LNCS 1123, pages 718–731. Springer-Verlag, 1996.

Tore A. Bratvold. *Skeleton-Based Parallelisation of Functional Programs.* PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 1994.

Silvia Breitinger, Ulrike Klusik, and Rita Loogen. An Implementation of Eden on top of Concurrent Haskell. In Werner Kluge, editor, *Implementation of Functional Languages (IFL'96)*, LNCS 1268, pages 142–161. Springer-Verlag, 1997.

David J. Busvine. *Detecting Parallel Structures in Functional Programs.* PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 1993.

Silvia Ciarpaglini, Marco Danelutto, Laura Folchi, Carlo Manconi, and Susanna Pelagatti. ANACLETO: A template-based p3l compiler. In *Proc. 7th Parallel Computing Workshop (PCW'97)*, pages P2–F–1–7. Australian National University, 1997.

Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

Jean-François Collard. Automatic parallelization of `while`-loops using speculative execution. *Int. J. Parallel Programming*, 23(2):191–219, 1995.

Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symp. on Principles of Programming Languages (POPL'82)*, pages 207–212. ACM Press, 1982.

John Darlington, Anthony Field, Peter Harrison, Paul Kelly, David Sharp, Qian Wu, and Ronald L. While. Parallel programming using skeleton functions. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *Parallel Architectures and Languages Europe (PARLE '93)*, LNCS 694, pages 146–160. Springer-Verlag, 1993.

John Darlington, Yi-ke Guo, Hing Wing To, and Jin Yang. Skeletons for structured parallel composition. In *Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 19–28. ACM Press, 1995.

Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(2):155–173, 1982.

Luis A. Galán, Cristóbal Pareja, and Ricardo Peña. Functional skeletons generate process topologies in Eden. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs (PLILP'96)*, LNCS 1140, pages 289–303. Springer-Verlag, 1996.

Sergei Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96, Vol. II*, LNCS 1124, pages 401–408. Springer-Verlag, 1996.

Sergei Gorlatch and Susanna Pelagatti. A transformational framework for skeletal programs: Overview and case study. In José D. P. Rolim et al., editor, *Parallel and Distributed Processing*, LNCS 1586, pages 123–137. Springer-Verlag, 1999. IPPS/SPDP'99 Workshops.

Martin Griebl and Christian Lengauer. On the space-time mapping of WHILE-loops. *Parallel Processing Letters*, 4(3):221–232, September 1994.

Haskell 98–Report. Simon L. Peyton Jones and John Hughes, editors. Haskell 98: A non-strict, purely functional language. Technical report, http://haskell.org, 1999.

$\mathcal{HDC}$ website, Lehrstuhl für Programmierung, Universität Passau. The $\mathcal{HDC}$ compiler project. http://www.fmi.uni-passau.de/cl/hdc/, 1999.

Christoph A. Herrmann and Christian Lengauer. On the space-time mapping of a class of divide-and-conquer recursions. *Parallel Processing Letters*, 6(4):525–537, 1996.

Christoph A. Herrmann and Christian Lengauer. Parallelization of divide-and-conquer by translation to nested loops. Technical Report MIP-9705, Fakultät für Mathematik und Informatik, Universität Passau, March 1997.

Christoph A. Herrmann and Christian Lengauer. Size inference of nested lists in functional programs. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Proc. 10th Int. Workshop on the Implementation of Functional Languages (IFL'98)*, pages 346–364. Department of Computer Science, University College London, 1998.

Christoph A. Herrmann and Christian Lengauer. Parallelization of divide-and-conquer by translation to nested loops. *J. Functional Programming*, 1999. To appear.

Zhenjiang Hu. Personal communication at the Dagstuhl Seminar on High-Level Parallel Programming, April 1999.

Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA'85)*, LNCS 201. Springer-Verlag, 1985.

Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. A new approach to inlining. Technical Report 92/06, Computer Science Department, SUNY at Stony Brook, 1992.

Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. On the conversion of indirect to direct recursion. *ACM Letters on Programming Languages and Systems*, 2(1–4):151–164, 1993.

Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

Björn Lisper. Data parallelism and functional programming. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, LNCS 1132, pages 220–251. Springer-Verlag, 1996.

Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. on Programming Languages and Systems*, 4(2):258–282, April 1982.

Jayadev Misra. Powerlist: A structure for parallel recursion. *ACM Trans. on Programming Languages and Systems*, 16(6):1737–1767, November 1994.

Marian Musiol. Implementierung von parallelem Divide-and-Conquer auf Gittertopologien. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1996.

Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

Lawrence G. Paulson. *ML for the Working Programmer.* Cambridge University Press, 2 edition, 1996.

Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall Int., 1987.

Simon L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In Hanne Riis Nielson, editor, *Programming Languages and Systems (ESOP'96)*, LNCS 1058, pages 18–44. Springer-Verlag, 1996.

Michael J. Quinn. *Parallel Computing.* McGraw-Hill, 1994.

Christian Schaller. Elimination von Funktionen höherer Ordnung in Haskell-Programmen. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, September 1998.

Stephen K. Skedzielewski. Sisal. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 4. ACM Press, 1991.

Hannu Toivonen. *Discovery of Frequent Patterns in Large Data Collections.* PhD thesis, Department of Computer Science, University of Helsinki, 1996.

Phil W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + strategy = parallelism. *J. Functional Programming*, 8(1):23–80, January 1998.

Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1995.

Andreas Zeller and Dorothea Lütkehaus. DDD – a free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, January 1996.