Using Metaprogramming to Parallelize Functional Specifications

Christoph A. Herrmann and Christian Lengauer

University of Passau, Germany

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

Abstract. Metaprogramming is a paradigm for enhancing a general-purpose programming language with features catering for a special-purpose application domain, without a need for a reimplementation of the language. In a staged compilation, the special-purpose features are translated and optimized by a domain-specific preprocessor, which hands over to the general-purpose compiler for translation of the domain-independent part of the program. The domain we work in is high-performance parallel computing. We use metaprogramming to enhance the functional language Haskell with features for the efficient, parallel implementation of certain computational patterns, called skeletons.

1 Introduction

We present work in progress on the high-level construction of parallel programs. Our aim is to organize the computations statically, as far as possible, i.e., to assign each operation to a particular processor and time step, and place all required communications. To remain sufficiently flexible, we must be able to gather and exploit information about values appearing at run time. We distinguish between two kinds of run-time values: those which are useful for controlling the parallelization and those which are not. The useful values carry information about temporal and spatial extent of the computation, as well as the values of iteration variables and the sizes of data structures. For instance, the length of a list can be made explicit by a suitable, non-standard source-level representation [16]. If the actual length is not available at compile time, a structural parameter is used to represent it.

We see our work as a contribution to the area of *domain-specific program generation and optimization* – specifically, for the domain of high-performance parallel computing, but with a relevance for other domains as well. Application programmers in the domain should have the comfort of a relatively abstract programming style, which unburdens them from the consideration of domain-specific implementational details. Expert programmers in the domain should have all means at their disposal for exploiting domain-specific knowledge in an optimizing compilation of application programs.

Concerning our specific domain, the two most common ways of infusing high-performance parallelism into programs today are:

– implicitly, by parallelizing an imperative program automatically and
– explicitly, by augmenting it with calls of modules in a run-time library like MPI [18] which implements interprocess operations.

Automatically parallelized programs often lack efficiency due to inherent difficulties in a precise dependence analysis, whereas the explicit programming of parallelism burdens the programmer with additional considerations of low-level details like memory organization, buffering, marshaling data structures for communication, synchronization, etc.

The community of high-level parallel programming has been pursuing a third approach based on program templates, so-called *skeletons* [1, 4–7, 11, 19]. A skeleton provides an abstract interface of a programming paradigm and encapsulates the concrete implementational details. Skeletons can represent simple schemata, like vector parallelism or pipelining, and also more complex schemata like divide-and-conquer or branch-and-bound. In our own previous work, we have identified a skeleton in a functional program by a specially named higher-order function and have used a program generator to produce the parallel implementations of skeletons, specialized for a particular call context [15]. The parallel code structure is specified by the skeleton designer, using his/her expert knowledge, and filled in with sequential code parts, specified as skeleton parameters. These parts are translated by a compiler independently from the domain. However, the compiler must be tailored for the integration of skeletons. In our experience, engineering difficulties result from a tight connection between the code generator for skeletons and the compiler, concerning calling conventions, memory organization, etc. [12, 14].

Here, we outline a more flexible approach which enables us to use a general-purpose compiler for the translation of the domain-independent part of the program. For convenience, we use the functional language Haskell but, in principle, this approach should carry over to imperative programming languages.

The compilation process consists of two phases, of which the first constructs an interface between the application program and the skeletons it calls and the second organizes the parallelism. The following section sketches the first, the rest of the paper concentrates on the second phase. We do not consider source programs as the object of our discussion. Instead, we show how to construct target programs bottom up by starting with atomic operations in the target language and combining smaller programs to larger programs, via sequential and parallel composition. This compositional approach motivates the use of algebraic specifications in a functional language. We define the abstract syntax of the coordination language by an algebraic data type in Haskell. Its semantics is defined by a Haskell function.

Section 3 introduces our language of symbolic expressions which enables an index-based description of sets of parallel tasks or sequences of tasks, where the number of tasks need not be a constant but can depend on structural parameters or iteration variables. Section 4 introduces the formal specification of parallel computations, by way of three examples. Section 5 defines the semantics of these structures. How program properties can be calculated is explained in Section 6. Section 7 is about the generation of parallel code. In Section 8, we summarize and propose our plans for future work.

2 The Interface Between Compiler and Skeletons

In contrast to our previous approach in the $\mathcal{HDC}$ project [15], the individual requirements of skeletons are no longer of concern to the compiler. Instead, the domain-specific side is represented by an algebraic data type which we make powerful enough to express our skeletons.

2.1 Example 1: Parallel `map`

Figure 1 depicts the parallelization of function `map` which takes a customizing function `f` and a list `xs` and applies `f` to each element of `xs`, yielding a list `ys`. The domain-independent parts of the program are the internals of `f` and `xs`, domain-specific is the implementation of `map` and the mechanisms of dealing with its parameters and results: since the used data types – lists and functions– are inappropriate for communication, we insert adaptation functions (see the middle part of the figure).

In a specialization step, `map` is being replaced by a function `mapIntVector`, which operates on integer vectors only. Function `f` is encoded by a data structure which carries all necessary context information to work in isolation, without access to the heap in which it has been created. In contrast to a global defunctionalization [2], as used –in a relaxed form–

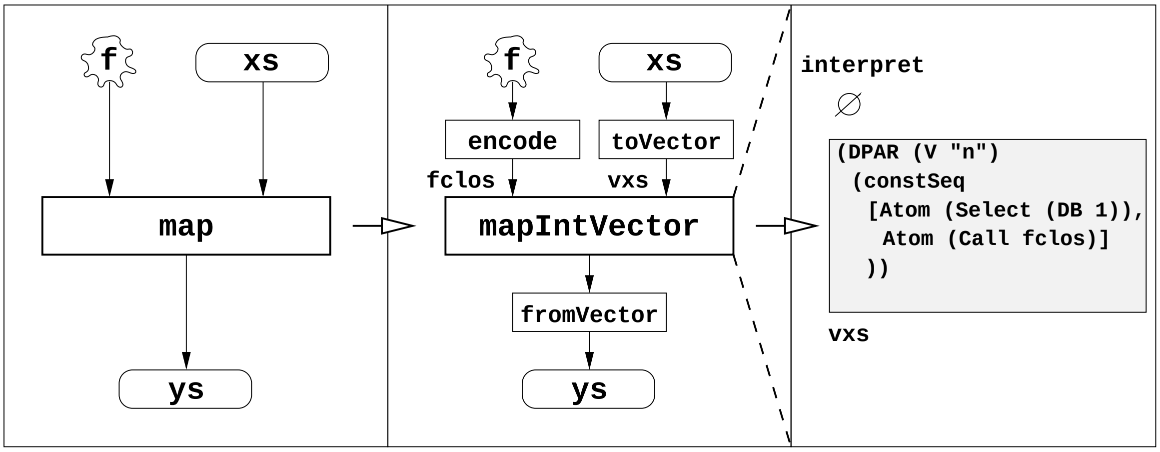

Fig. 1. Interface between compiler and skeletons

in the $\mathcal{HDC}$ compiler [14], we perform this encoding only locally. We assume that program analysis reveals that the use of `map` is monomorphic here.

Function `mapIntVector` can then be replaced by an interpretation of a structural description of how a parallel `map` works (right compartment of the figure). While the encoded customizing function `f` is called within this description, the input vector is given as a third argument to the interpretation. For simplicity, we refer to the length of this vector by a structural parameter `"n"` in the description. The first argument of the interpretation function is an empty environment of iteration variables.

In a compilation, the interpretation function will be replaced by the implementation of the parallel structure. The inputs and outputs of `mapIntVector` have to be converted by means of a standard Haskell-to-C interface.

3 Symbolic Expressions

The static part of the parallelization of the program may depend on symbolic expressions in multiple ways, e.g., the degree of parallelism can depend on the following:

– The length of an input vector. In this case, we introduce the length as a structural parameter which appears in expressions dealing with indexing, communication partners, etc.
– The iteration count in the execution of a sequential loop. Here, we establish compile-time access to the (virtual) iteration variable using the de Bruijn indexing schema [10].

Symbolic expressions are members of an algebraic data type `SymExp`, which should have at least the following constructors:

1. `C Rational`: a constant number. Final results of summations which form counts of entities are always integral, but non-integral numbers can appear in intermediate results of closed forms.
2. `V String`: a structural parameter.
3. `DB Int`: a de Bruijn index, i.e., a natural number specifying how many levels in a nested variable binding one has to traverse to reach the binding level of the variable in question. De Bruijn indices provide a concise means of dealing with (indexing) functions in the context-free language `SymExp`.[1] These indices represent both quantified variables in the

[1] Note that `SymExp` aims to be a short internal representation language; a specification language given to a programmer should provide meaningful variable names.

symbolic expressions (see `Sum` below) and iteration variables of the loop body in which the symbolic expression appears.[2]
4. `Op`: a binary operation, as in arithmetic or boolean expressions.
5. `Quant` *opcode* n e: an aggregated application of an associative operation *opcode*. This quantification is meant to be eliminated in symbolical simplifications. At present, we include only `Add` and `Max`. E.g., with *opcode*=`Add`, the quantification specifies $\sum_{i=0}^{n-1} e_i$, where i is a fresh variable and e_i is obtained from e by replacing at each binding level j all indices `DB` j by `C` i. Both n and e are symbolic expressions. `Quant` is used as an intermediate form and must be either resolved at compile time[3] or replaced by a loop which computes it at run time.
6. `Case` n xs: specifies a case distinction. The symbolic expression n determines the natural number of the case, xs is a list of symbolic expressions, of which element n is selected as the value of the entire case expression.

```
data SymExp = C     Rational                — constant
            | V     String                  — structural parameter
            | DB    Int                     — de Bruijn index
            | Op    OpCode SymExp SymExp    — combining operation
            | Quant OpCode SymExp SymExp    — aggregated operation
            | Case  SymExp [SymExp]         — case distinction
```

The data type `OpCode` comprises the names of all binary operations we use and may be extended appropriately. It is also used in contexts other than in `SymExp`, e.g., in the type `FunSymbols` presented later on.

```
data OpCode = Add | Sub | Mul | Pow | Min | Max | IsBitSet | BitExOr
```

`IsBitSet` applied to i and n delivers the value of bit i in the binary representation of n, where 0 is the index of the least significant bit. `BitExOr` computes the bitwise exclusive or of two integers.

Symbolic expressions can be used inside an environment of symbolic expressions, named `SymEnv`:

```
newtype SymEnv = SymEnv [SymExp]
```

The empty symbolic environment is denoted ∅:

```
∅ :: SymEnv
∅ = SymEnv []
```

An element is added to a symbolic environment with ◁. When doing so, de Bruijn indices appearing in the environment have to be incremented by function `incDB`.

```
(◁) :: [SymExp] → SymEnv → SymEnv
xs ◁ (SymEnv env) = SymEnv (xs ++ map (incDB (length xs)) env)
```

`incDB` traverses a symbolic expression and increases each de Bruijn index by `k`.

```
incDB :: Int → SymExp → SymExp
incDB k (DB i) = DB (i+k)
incDB k (Op opcode a b)    = Op    opcode (incDB k a) (incDB k b)
incDB k (Quant opCode a b) = Quant opCode (incDB k a) (incDB k b)
incDB k (Case c xs)        = Case (incDB k c) (map (incDB k) xs)
incDB k x = x
```

Function `indEnv` accessed positions in the environment.

```
indEnv :: SymEnv → Int → SymExp
indEnv (SymEnv env) i = env !! i
```

[2] Of course, the latter carry the higher numbers.
[3] Simple summations can be expressed in closed form by a polynomial.

4 Specification of Static Parallel Computation Structures

4.1 The Language

The entire program is inductively composed of atomic functions, and by sequential composition, parallel composition and deterministic choice of smaller programs.

In our model, we represent each of these entities by a constructor of an algebraic data type `S` in Haskell. `S` is parameterized in the type variable `a`, which represents the union of all types that are used in the application program.

```
data S a = Atom a
         | Seq SymExp (S a)
         | Alt SymExp [S a]
         | DPar SymExp (S a)
         | ParComm { stages :: SymExp, parts :: SymExp, elemFun :: S a,
                     noInputs :: SymExp, sources :: SymExp  }
```

The constructors of type `S` have the following meaning:

1. `Atom`: a function which is not parallelized. Each function block takes one argument and delivers one result. Multiple arguments and results have to be represented by an appropriate data structure; we use only vectors of restricted integers here.
2. `Seq` n e: a sequence of functions. Parameter n specifies the length of the sequence and e an indexed expression which specifies a particular element in the sequence if, at nesting level i, `DB` i is replaced by the current index in the sequence.
3. `Alt` i *alts*: a deterministic choice. Parameter i is a symbolic expression which constitutes an index. Parameter *alts* is a list of alternatives. If the value of i is not negative and falls within the length of the list, element i of *alts* is selected. Otherwise, the last element of *alts* is selected.
   Intentionally, we permit case distinctions which influence the structure of parallelism to depend on structural parameters only. The justification for this restriction is to be able to calculate the allocation and communications at compile time.
4. `DPar` n e: a parallel composition of disjoint tasks, i.e., tasks which do not communicate with each other. Simplified forms of divide-and-conquer can be described with this constructor, e.g., the forms `dcA`, `dcB` and `dcC` in [13]. Parameter n specifies the number of parallel tasks, the second parameter e part p, if at level i each occurrence of `DB` i in e is replaced by p.
   Each tasks computes a private result vector. All vectors are then concatenated to form the result of the entire `DPar` expression, following our convention that each block takes a single input and a single output object.
5. `ParComm`: specifies parallel tasks which communicate with each other. The explanation of this quite sophisticated constructor is deferred to Subsection 4.5.

In order to keep the set of constructors minimal, there are no separate constructors for sequential or parallel compositions of a fixed size. These are expressed in terms of `Seq`/`DPar` and `Alt`, by the following functions:

```
constSeq, constDPar :: [S a] → S a
constSeq  xs = Seq  (C (fromIntegral (length xs))) (Alt (DB 0) xs)
constDPar xs = DPar (C (fromIntegral (length xs))) (Alt (DB 0) xs)
```

4.2 The Domain of Values

From a programming language design perspective, a richer type system would be desirable but, to keep our exposition brief, we make the input and output of each atomic function uniformly a vector (or, in Haskell, a list) of 32-bit integers. Thus, the booleans are represented by integers (0 for `False` and 1 for `True`) and nested vectors have to be flattened. In a sequential composition, the output of stage i becomes the input of stage $i+1$. In a parallel composition, the input is broadcasted to all parts and the result vectors are concatenated to form the entire result.

The atomic functions are built using language `FunSymbols`:

```
data FunSymbols = FOp OpCode
                | Select SymExp
                | Id
                | IsNeg | Negate | Square
                | IfThenElse FunSymbols FunSymbols FunSymbols
                | Tuple [FunSymbols]
                          ⋮
```

- `FOp` *opcode* combines the first two input vector elements by function *opcode* and produces a single element vector.
- `Select` i selects element i of the input vector.
- `Id` delivers the input vector as a result.
- `IsNeg` delivers `True` (`[1]`) iff the first input element is a negative number.
- `Negate` delivers the negation of the first input element
- `Square` squares the first input element.
- `IfThenElse` c x y applies c; if the result of c is `True` it applies x, otherwise y.
- `Tuple` promotes a list of atomic functions to a new atomic function which produces the concatenation of all individual results.

4.3 Example 1, Revisited

Consider the right compartment of Fig. 1, which depicts a parallel application of function `map`. Here, we use it to square each element of a list. Its representation in our formalism is

```
ex1:: S FunSymbols
ex1 = DPar (V "n") (constSeq [Atom (Select (DB 1)), Atom Square])
```

`DPar (V "n")` tells us that there are `n` tasks to be combined in parallel. The input vector is broadcasted to each task. Each task performs two atomic operations in sequence (`constSeq`):

1. `Select (DB 1)` selects the element whose index equals the number of the task.[4]
2. `Square` squares this number.

Each parallel task delivers a vector of length 1. The entire result is the concatenation of all of these, i.e., a vector of length `n`.

4.4 Example 2: Parallel Power Computations

Fig. 2 depicts the (synthetic) example of the computation of the following Haskell function `f_ex2`:

```
f_ex2 :: [Int] → [Int]
f_ex2 xs = let x = xs !! 0 + xs !! 1
               y = if x<0 then (−x)^3 else x^2
           in [y]
```

Three parallel tasks are produced, the first evaluates `x<0`, the second `(-x)^3` and the third `x^2`.

In our formalism, the parallel computation is expressed as follows:

[4] Both `ParSeq` and `constSeq` (via `Seq`) introduce a new iteration variable. Thus, `DB 0` would refer to the index in the sequence.

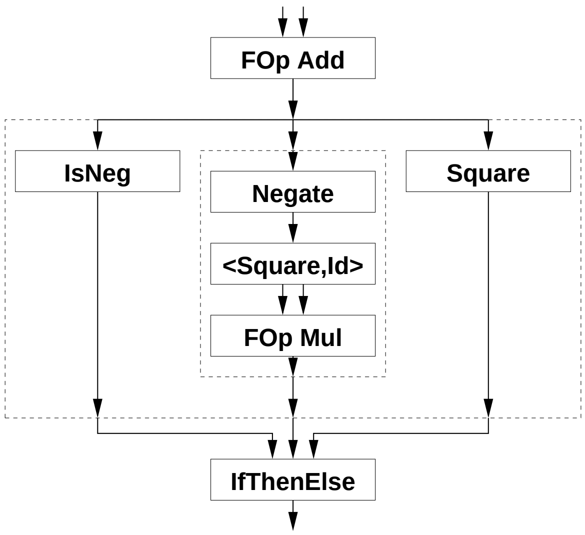

Fig. 2. Power example

```
ex2 :: S FunSymbols
ex2 = constSeq
        [Atom (FOp Add),
         constDPar [Atom IsNeg,
                    constSeq [Atom Negate,
                              Atom (Tuple [Square,Id]),
                              Atom (FOp Mul)],
                    Atom Square],
         Atom $ IfThenElse (Select $ C 0) (Select $ C 1) (Select $ C 2)]
```

4.5 Communicating Parallel Tasks

The structure language above permits us to define a structured task-parallel schema, in contrast to the more error-prone fork/join parallelism. There are a number of efficient parallel computation schemata, e.g., the butterfly schema, which require multiple exchanges of data between processes. This can be done safely, if the communicating processes know each other in advance and exchanges happen at defined synchronization points in the program. In the most simple form, this is the case in data parallelism, which can be implemented by an alternation of phase of simple local computation and a phase of global (all-to-all) data exchange. More complex models are communication-closed layers [8] and the BSP model.

Even more flexibility can be achieved with our principle of so-called control-closed blocks [14], in which communication-closed layers are (1) only required for a subset of processes which do not communicate to the outside at the same time and (2) the processes inside each layer can be partitioned into subblocks which perform the same schema independently.

Like in the case of `DPar` for `ParComm`, the input of a block is input to all subblocks resp. atomic elements (e.g., by a broadcast) and the output of a block is formed by concatenating the output lists of all subblocks. The difference is that there can be communications between tasks, by synchronization of all processors of the block. The arguments of the `ParComm` constructor are:

1. `stages :: SymExp`: the number of levels of computation which are finished by a synchronization and exchange of data of the processors of the block.

2. `parts :: SymExp`: the number of parallel tasks which always equals the number of subblocks the block is divided into.
3. `elemFun :: S a`: the function each of the parallel tasks performs at each stage. At the outermost level, the index of the stage is accessible by `DB 0` and the index of the task by `DB 1`.
4. `noInputs :: SymExp`: the number of subblocks of the previous stages used to obtain the input of each `elemFun`, expressed in the same two indices as in 3.
5. `sources :: SymExp`: those parts of the previous stage whose result vectors are concatenated to form the input vector of each `elemFun`, where the current stage is given by `DB 0`, the own part by `DB 1` and the index of the subblock from which the input is taken by `DB 2`.

At stage 0, `noInputs` and `sources` are undefined.

4.6 Example 3: Bitonic Sort

Let us take the butterfly computation schema, depicted in Fig. 3, as an example of the use of `ParComm`. At stage s of $n+1$ stages (stage 0 does not perform exchange or computation), each processing element with index i receives data from the processing element with index (i `BitExOr` 2^{n-s}).

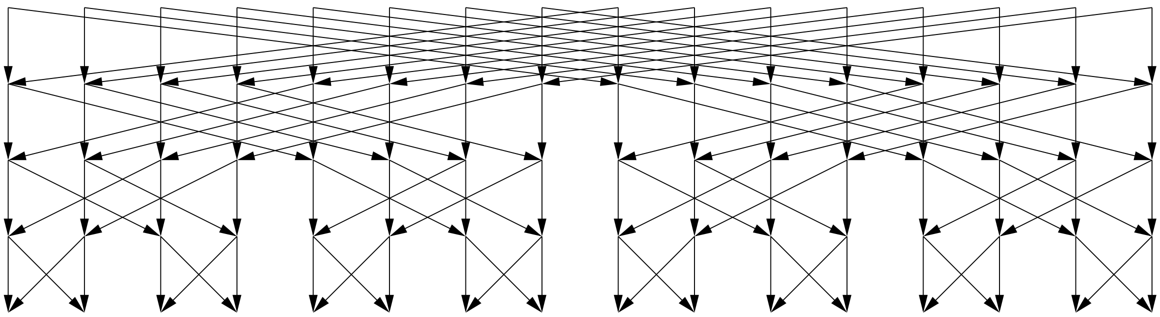

Fig. 3. Butterfly (n=4)

The butterfly schema can be formally specified by function `ex3`, with the element function customized such that the entire function of the structure computes the bitonic sort [17]:

```
ex3 n = ParComm (C (n+1))
                (Op Pow (C 2) (C n))          — 2^n processes
                (Alt (DB 0)
                     [Atom (Select (DB 1)),
                      Alt (Op IsBitSet (Op Sub (C n) (DB 0)) (DB 1))
                          [Atom (FOp Min), Atom (FOp Max)]])

                (C 2)                        — each element has 2 inputs
                (Case (DB 2)                 — sources of inputs
                      [DB 1, Op BitExOr (DB 1)
                                 (Op Pow (C 2) (Op Sub (C n) (DB 0)))])
```

5 Interpretation

Our interpreter of parallel structures is meant to define the semantics of our language and to be used in the development of parallel designs and in extensive tests of the generated code.

5.1 An Interface to Types of the Application Domain

The following class definition connects the Haskell language with the union `a` of all data types of the application domain. This facilitates the definition of the interpreter without knowledge of the actual application domain.

```
class AppDomain a where
 iter      :: SymEnv → (a → SymExp → a) → a → SymExp → a
 mkList    :: SymEnv →      (SymExp → a)      → SymExp → a
 indexList :: SymEnv → a → SymExp → a
 funEval   :: SymEnv → FunSymbols → a → a
```

All class functions take an environment as a first argument.

- `iter` e f x n iterates function f n times, with initial value x, providing the current iteration count to each application of f.
- `mkList` e f n constructs a *list* (e.g., vector) in the language `a` of the application domain, i.e., the result is of type `a`, not `[a]`. n is the length of the list and f maps indices to list elements.
- `indexList` e xs i takes a *list* (e.g., vector) xs in the language `a` and an index i and delivers element i of xs.
- `funEval` e f x applies an atomic function f to x.

5.2 The Interpretation Function

Additionally, the interpreter requires three auxiliary functions:

- `getInt :: SymEnv -> SymExp -> Int`
  `getInt` e s evaluates the symbolic expression s to a Haskell `Int` value.
- `cI :: Int -> SymExp`
  `cI` converts a Haskell `Int` value into a symbolic expression.
- `convertLAD :: AppDomain a => [a] -> a`
  `convertLAD` converts a Haskell list into the list type of the application domain `a`, using function `mkList` of the class definition `AppDomain`.

Function `interpret` takes as arguments an environment of indices, the structure to be interpreted and an input value:

```
interpret :: AppDomain a ⇒ SymEnv → S FunSymbols → a → a
interpret env (Atom f) x = funEval env f x
interpret env (Seq n f) x  = iter env (\a i → interpret ([i] ◁ env) f a) x n
interpret env (Alt i fs) x = let l = length fs
                                 c = getInt env i
                             in interpret env (fs !! (if  (c ≥ 0 && c<l−1)
                                                          then c else (l−1))) x
interpret env (DPar n f) x  = mkList env (\i → interpret ([i] ◁ env) f x) n
interpret env a@(ParComm {}) x
 = let sts = getInt env $ stages a
       bls = getInt env $ parts a
       inputs i j xs = convertLAD
                     [ xs !! getInt  ([cI i, cI j, cI s] ◁ env) (sources a)
                     | s←[0..getInt ([cI i, cI j] ◁ env) (noInputs a)−1] ]
       stageI xs i = [ interpret  ([cI i, cI j] ◁ env) (elemFun a) (inputs i j xs)
                   | j ← [0..bls−1] ]
   in convertLAD $ foldl stageI (take bls (repeat x)) [0..sts−1]
```

5.3 Application in Our Example Domain

In our example domain, all data objects are vectors (lists in Haskell) of restricted integers, which can be communicated easily in a parallel setting. Vectors are used to pass multiple arguments to functions and also to store temporary values in atomic computations. Booleans are encoded by the integers 0 (for `False`) and 1 (for `True`). Composed data structures, as far as required for our examples, are flattened. Possibly, descriptors are required to restore the structure.

```
data IntVec = IV { unIV :: [Int] }
```

After making `IntVec` an instance of class `AppDomain`, we can apply the interpreter to our example programs:

1. `map` example:
   Since we have to resolve the structural parameter `n` in the interpretation, we modify `ex1` to a specification `ex1'` which uses a de Bruijn index instead.

   ```
   ex1' :: S FunSymbols
   ex1' = DPar (DB 0) (constSeq [Atom (Select (DB 1)), Atom Square])
   ```

   ```
   — Main> interpret ([C 4] ◁ ∅) ex1' (IV [1..10])
   — IV{unIV=[1,4,9,16]}
   ```

2. Power example:

   ```
   — Main> interpret ∅ ex2 (IV ([1,1]::[Int]))
   — IV{unIV=[4]}
   — Main> interpret ∅ ex2 (IV ([1,−3]::[Int]))
   — IV{unIV=[8]}
   ```

3. Bitonic sort example:

   ```
   — Main> interpret ∅ (ex3 4) (IV [1,3,5,7,9,11,13,15,18,16,14,12,10,8,6,4])
   — IV{unIV=[1,3,4,5,6,7,8,9,10,11,12,13,14,15,16,18]}
   ```

6 Calculating Program Properties

An important aspect of programming with parallelism is the analysis of program properties, i.e., the number of operations, the free schedule, the degree of parallelism, etc. Calculated for the entire program, these properties provide us with information about computation time, resources and efficiency. Calculated for parts of the program, they help to identify a suitable distribution of tasks across the parallel processors during code generation.

We calculate three program properties by a single compositional analysis, parameterized by the particular property. Function `work` computes the number of atomic computations, `depth` the length of the longest path and `usedPEs` the number of (virtual) processing elements required if each operation is executed following the free schedule (i.e., as soon as possible). The properties `work` and `depth` have been adopted from the language NESL [3].

```
data WDU = Work | Depth | UsedPEs

work, depth, usedPEs :: SymEnv → S a → SymExp
work    = wdu Work
depth   = wdu Depth
usedPEs = wdu UsedPEs

wdu :: WDU → SymEnv → S a → SymExp
```

For atomic functions, all three properties have the value 1.

```
wdu c env (Atom _) = C 1
```

Calculation of an alternative is done by applying the calculation to each case and reducing the selection to a case expression in the language of symbolic expressions.

```
wdu c env (Alt n fs) = simplify env
                       (Case n (map (wdu c env) fs))
```

Calculation of a sequential or parallel composition is performed by a quantification over the stages/parts, where the quantified expression is the property expression of the particular stage/part which is expressed in terms of the position of the stage/part. This position is accessed by `DB 0`. In a sequential composition, `work` and `depth` of the stages are added, while `usedPEs` is given by the maximum. In a parallel composition, `work` and `usedPEs` of all tasks are added, while `depth` is given by the maximum.

```
wdu c env (Seq n f) = simplify env
                      (Quant (case c of
                               Work    → Add
                               Depth   → Add
                               UsedPEs → Max) n (wdu c ([DB 0] ◃ env) f))
wdu c env (DPar n f) = simplify env
                      (Quant (case c of
                               Work    → Add
                               Depth   → Max
                               UsedPEs → Add) n (wdu c ([DB 0] ◃ env) f))
```

More complicated is the case of a parallel composition in which processes can communicate with each other at certain common synchronization barriers. This has the consequence that, for the calculation of the depth, we have to sum the maximum depths of all stages.

In order to simplify the allocation, we opted not to rebalance the load at each stage. As a consequence, we obtain vertical barriers between parts in addition to the horizontal barriers introduced by synchronization. This has the consequence that, for `usedPEs`, we have to sum up the maximum value of `usedPEs` for each parallel slice.

We observe that both `depth` and `usedPEs` require a sum of maximum, but, for the `depth`, the maximum is on the parts and the sum on the stages, where for `usedPEs`, the opposite is the case. This occurs when assigning the quantified range (`p` for parts, `s` for stages) to `n0` and `n1` and the de Bruijn index for the current stage (`dBs`) and the current part (`dBp`).

```
wdu c env (ParComm {stages=s,parts=p,elemFun=f})
        = let (op0,n0,dBs,op1,n1,dBp) = case c of
                                          Work    → (Add,s,1,Add,p,0)
                                          Depth   → (Add,s,1,Max,p,0)
                                          UsedPEs → (Add,p,0,Max,s,1)
          in simplify env $
                (Quant op0 n0
                   (Quant op1 (incDB 1 n1)
                              (wdu c ([DB dBs, DB dBp] ◃ env) f)))
```

Now, we can apply the calculation to our examples:

1. `map` example:

```
-- Main> [ f ∅ ex1 | f ← [work,depth,usedPEs] ]
-- (Quant Add (V "n") (C (2 ÷ 1)),Quant Max (V "n") (C (2 ÷ 1)),
--  Quant Add (V "n") (C (1 ÷ 1)))
-- Main> [ f ([C 4] ◃ ∅) ex1' | f ←[work,depth,usedPEs] ]
-- (C (8 ÷ 1),C (2 ÷ 1),C (4 ÷ 1))
```

2. Power example:

```
-- Main> [ f ∅ ex2 | f ← [work,depth,usedPEs] ]
-- (C (7 ÷ 1),C (5 ÷ 1),C (3 ÷ 1))
```

3. Bitonic sort example:

```
— Main> [ f ∅ (ex3 4) | f ← [work,depth,usedPEs] ]
— (C (80 ÷ 1),C (5 ÷ 1),C (16 ÷ 1))
```

7 Code Generation

In code generation, we distinguish two different approaches. The first is to compile the program, or a part of it, for a particular instantiation of all structural parameters. The second is to compile without instantiation.

1. If all structural parameters are instantiated, the specification may still contain symbolic expressions which cannot be resolved. However, with an abstract interpretation of the specification we can obtain a task graph. Via linear programming, we can compute from this task graph a space-time mapping which is globally optimal with respect to execution time, if given the number of processors. This can pay off frequently in practice, e.g., in chip design or the programming of device drivers.
2. If structural parameters remain, one cannot hope to calculate the optimal space-time mapping, since it depends on choices based on the values of these parameters and the occurring predicates are generally undecidable. Thus, code generation must remain flexible. If a condition cannot be checked at compile time but its knowledge would be useful, an `Alt` construct can be inserted which prescribes an alternative parallelization, in dependence of run-time values. In some cases, this `Alt` construct can be eliminated later by program specialization. We explain this type of code generation in the SPMD (single-program-multiple-data) format in more detail in the rest of the section.

7.1 Compilation to SPMD Code

Referring to the principle of control-closed blocks, each block –be it composed via `Seq` or `DPar`, resp. `ParComm`– is assigned a set of (virtual) processors for exclusive use. There is no communication of a block's set of processors to the outside other than at the block's beginning and end. Each block has a temporal master processor for (1) receiving input and sending output from the outside, (2) partitioning the block into subblocks and (3) distributing work and receiving results of the subblocks' masters.

Compilation is carried out recursively for each block. A code frame for each kind of block is created, code parts for the subexpressions are created and these parts are inserted into the code frame. In particular, this frame looks as follows:

– `Atom`: compute only if processor is block master.
– `Seq n sub`: create a sequential loop with `n` iterations.
– `DPar` ...: divide the block into subblocks. The details are explained in Section 7.2.
– `ParComm` ...: divide the block into subblocks. The details are explained in Section 7.3
– `Alt n cases`: create a switch statement.

7.2 Compilation of DPar

Tasks are mapped to blocks of natural numbers which represent identifiers of virtual processors. Each processor calculates the number of subblocks which precede it in the surrounding block. This is done by recursively calculating the number of used processors of a sufficient prefix of the list of parallel tasks, using function `usedPEs`. During this computation, the master processors of the preceding subblocks and of the own sublock are recorded.

Then, the code for the subtask is generated. If the processor is the master of its own subblock, it has to receive a task from the processor of the surrounding block before and to send it back the result afterwards. Otherwise, it does not perform communication at this level.

7.3 Compilation of ParComm

Similarly to the case of disjoint parallelism, each processor derives the extent of the part it belongs to. The difference is that the computation of the used processing elements of each preceding block must be computed as the maximum over all stages, whereas, in `DPar`, we only have a single stage. As in the case of `DPar`, the processor performs communications only at this nesting level if it is the master of its block.

Note that each task may consist of a block of processors, of which only one distinguished processor –the master of the block– participates in the communication at the respective level, while the other processors of the block can be used for a nested parallel subcomputation. This is similar to the concept of an MPI intercommunicator [18].

In the case that a master performs communication, the number of receives and their sources can be obtained by evaluating `noInputs` and `sources`. There are several options of carrying out the communication; all of them require a computation of processor IDs of the masters of the other parts.

1. Each processor evaluates the sources of the other masters in order to determine whether it has to perform a send operation to a particular processors. This may be best if the evaluation at compile time results in a simple expression; otherwise it may incur a high administrative overhead.
2. No send operations are performed but the receivers access their input by one-sided calls. Result values must either be recorded over all stages or protected by a barrier synchronization until they are read.
3. All master processors constitute an MPI group and perform an all-to-all communication in this group.

Case 1 is our favourite, since the coordination language is meant to define skeletons, for which compile-time knowledge of the (parameterized) communication pattern is crucial.

One should also consider the case that compile-time analysis might identify collective communications which are more efficient than all-to-all, e.g., broadcast, gather and scatter. Generally, collective communications are known to achieve higher efficiency than point-to-point communications [9].

8 Summary

In our previous work, the focus was on the compilation of a *fixed* source language ($\mathcal{HDC}$ in our case), allowing for external implementations of specific program parts, the skeletons, which have to be implemented directly in the target language (C+MPI in our case). This handled to several inconveniences:

- Since the connection between skeleton implementations and the compiler for the rest of the program is comparatively tight, we were forced to implement both sides, the domain-independent compiler and the framework for the domain-specific skeleton implementations.
- We implemented only a small subset of our base language Haskell. Also, we did not spend time reimplementing all the methods in code optimization which a Haskell compiler applies.

Here, we have presented a more comfortable and flexible option based on the paradigm of metaprogramming: the expert programmer can design skeleton implementations in a small, customized coordination language for which a special-purpose compiler has been crafted. Domain-independent portions of source code are encapsulated in atomic operations; we can reuse existing technology (compilers, programming tools, etc.) for their development.

Much of our technical exposition has been concerning structures which neither application programmer nor skeleton programmer will need to handle, e.g., the structure language `S` and the de Bruin indexing schema. These are used only for the internal representation. The coordination language will be more comfortable, e.g., allow meaningful variable names, but programs written in it will still be transformable to `S`.

In contrast to the work of many others in the parallelization community, we do not impose specific restrictions on our indexing expressions, e.g., affine linearity. This incurs the risk that index expressions might not be computable before run time. Simplification plays an important role in our parallelization. Symbolic expressions for which no solution algorithm is known are treated by heuristics, possibly collected in a library of patterns which is successively extended.

Acknowledgements

We are grateful to the members of our group for helpful remarks at a recent presentation of this work and, particularly, to Paul Feautrier who attended the presentation while visiting on a Procope exchange. Thanks also to the anonymous reviewers, especially for the suggestion to use type classes for generalization.

References

1. Bruno Bacci, Sergei Gorlatch, Christian Lengauer, and Susanna Pelagatti. Skeletons and transformations in an integrated parallel programming environment. In *Parallel Computing Technologies (PaCT-99)*, LNCS 1662, pages 13–27. Springer-Verlag, 1999.
2. Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. *SIGPLAN Notices*, 32(8):25–37, 1997. *Proc. ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'97).*
3. Guy Blelloch. NESL: A nested data-parallel language (Version 3.1). Technical Report CMU-CS-95-170, School of Computer Science, Carnegie-Mellon Univ., 1995.
4. G.H. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96: Parallel Processing, Vol. I*, LNCS 1123, pages 718–731. Springer-Verlag, 1996.
5. Tore A. Bratvold. *Skeleton-Based Parallelisation of Functional Programs.* PhD thesis, Dept. Computing and Electrical Engineering, Heriot-Watt Univ., 1994.
6. Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.
7. 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, *PARLE'93: Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science 694, pages 146–160. Springer-Verlag, 1993.
8. Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(2):155–173, 1982.
9. Sergei Gorlatch. Send-Recv considered harmful? Myths and truths about parallel programming. In Victor Malyshkin, editor, *Parallel Computing Technologies (PaCT 2001)*, Lecture Notes in Computer Science 2127, pages 243–257. Springer-Verlag, 2001.
10. Chris Hankin. *Lambda Calculi.* Oxford University Press, 1994.
11. 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.

12. Christoph A. Herrmann, Christian Lengauer, Robert Günz, Jan Laitenberger, and Christian Schaller. A compiler for $\mathcal{HDC}$. Technical Report MIP-9907, Fakultät für Mathematik und Informatik, Univ. Passau, May 1999.

13. Christoph A. Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions.* PhD thesis, Fakultät für Mathematik und Informatik, Univ. Passau, 2000. Logos-Verlag.

14. Christoph A. Herrmann and Christian Lengauer. The $\mathcal{HDC}$ compiler project. In Alain Darte, Georges-André Silber, and Yves Robert, editors, *Proc. Eighth Int. Workshop on Compilers for Parallel Computers (CPC 2000)*, pages 239–253. LIP, ENS Lyon, 2000.

15. Christoph A. Herrmann and Christian Lengauer. $\mathcal{HDC}$: A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2–3):239–250, 2001.

16. Christoph A. Herrmann and Christian Lengauer. A transformational approach which combines size inference and program optimization. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation (SAIG'01)*, Lecture Notes in Computer Science 2196, pages 199–218. Springer-Verlag, 2001.

17. Donald E. Knuth. *The Art of Computer Programming, Vol. 3: Searching and Sorting.* 2nd edition, 1998.

18. MPI Forum. *MPI: A Message-Passing Interface Standard.* Univ. of Tennessee at Knoxville, 1995.

19. David B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *J. Parallel and Distributed Computing*, 28:65–83, 1995.