Optimizing Space of Parallel Processes

Manfred Schmidt-Schauß*
Goethe-University
Frankfurt am Main
schauss@ki.cs.uni-frankfurt.de

Nils Dallmeyer*
Goethe-University
Frankfurt am Main
dallmeyer@ki.cs.uni-frankfurt.de

This paper describes a new approach to analyze and explore space-improvements in the concurrent and functional process-calculus CHF. Space-improvements are defined as a generalization from a deterministic pure functional language. The main part of the paper is a polynomial algorithm for space optimizations of parallel independent processes. Applications of this algorithm are: (i) affirmation of space improving transformations for particular classes of program transformations; (ii) support of an interpreter-based method for refuting space-improvements; and (iii) as a stand-alone offline-optimizer for space (or similar ressources) of parallel processes.

1 Introduction

The main motivation for this research is to investigate the space optimization and space improvements in concurrent languages. Our model is the functional process calculus CHF (Concurrent Haskell with Futures, see [1, 5, 6]) that permits pure functional modelling in combination with monadic execution of processes with synchronization. It provides mutable storage using one-place buffers (MVars) and it also permits starting subprocess as threads that return a value which is referenced by so-called futures. CHF is related to concurrent Haskell. We will employ a variant that also includes garbage collection, hence our analyses of space consumption, optimization and improvement assume that garbage is already removed. Related work on space improvements in deterministic call-by-need functional languages is [4, 3, 8], where concepts of space measurements and improvement are defined and many transformations are proven to be space improvements. The long-term goal is to transfer and extend the methods to parallel evaluated processes. Since this appears too ambitious, we first analyse parallel threads which are independent or have only rare interactions by a controllable form of synchronization.

The space behavior of threads that are evaluated in parallel can be described as follows. We assume that there is a shared memory, where the state of every process is stored. The space that a single thread may potentially use, or is relevant for the thread, subdivides into space that is only used by the thread and other data or functions that are used by several threads. Clearly, both kinds of storage are relevant for space analyses and for checking space improvements.

In the following we concentrate on the thread-only space and algorithms for optimizing it and leave the common storage for future analyses. One motivation for such an optimization analysis is to make a search for potential counterexamples to conjectures for space-improvements feasible. Even in the case of only two independent threads the computation of the minimally necessary (thread-local) space to run the threads leads to an exponential number of different schedules.

As we will see below, a deeper analysis shows that for independent processes (without communication), this minimum can be computed with an offline-algorithm in polynomial time (Theorem 4.27).

The model for processes is rather abstract insofar as it only models the thread-local space as a sequence of numbers. This simplicity invites applications of the space-optimization algorithm also for

*supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SCHM 986/11-1.

Submitted to:
WPTE18

$$
\begin{array}{rcl}
P \in \mathit{Proc} & ::= & (P_1 \mid P_2) \mid x \Leftarrow e \mid \nu x.P \mid x\,\mathbf{m}\,e \mid x\,\mathbf{m}- \mid x = e \mid \emptyset \\
e \in \mathit{Expr} & ::= & x \mid m \mid \lambda x.e \mid (e_1\ e_2) \mid \mathtt{seq}\ e_1\ e_2 \mid c\ e_1 \ldots e_{\mathrm{ar}(c)} \mid \mathtt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \mathtt{in}\ e \\
 & & \mid \mathtt{case}_T\ e\ \mathtt{of}\ (c_{T,1}\,x_1 \ldots x_{\mathrm{ar}(c_{T,1})} \to e_1) \ldots (c_{T,|T|}\,x_1 \ldots x_{\mathrm{ar}(c_{T,|T|})} \to e_{|T|}) \\
m \in \mathit{MExpr} & ::= & \mathtt{return}\,e \mid e \gg\!= e' \mid \mathtt{future}\,e \mid \mathtt{takeMVar}\,e \mid \mathtt{newMVar}\,e \mid \mathtt{putMVar}\,e\,e' \\
\tau \in \mathit{Typ} & ::= & \mathtt{IO}\ \tau \mid (T\ \tau_1 \ldots \tau_n) \mid \mathtt{MVar}\ \tau \mid \tau_1 \to \tau_2
\end{array}
$$

Figure 1: Syntax of expressions, processes, and types

$$
\begin{array}{lll}
\mathtt{size}(x) & = 0 & \mathtt{size}(e_1\ e_2) = 1 + \mathtt{size}(e_1) + \mathtt{size}(e_2) \\
\mathtt{size}(\lambda x.e) & = 1 + \mathtt{size}(e) & \mathtt{size}(\mathtt{case}\ e\ \mathtt{of}\ \mathtt{alt}_1 \ldots \mathtt{alt}_n) = 1 + \mathtt{size}(e) + \sum_{i=1}^{n} \mathtt{size}(\mathtt{alt}_i) \\
\mathtt{size}(P_1 \mid P_2) & = \sum \mathtt{size}(P_i) & \mathtt{size}((c\ x_1 \ldots x_n)\ \mathtt{->}\ e) = 1 + \mathtt{size}(e) \\
\mathtt{size}(x\ op\ e) & = 1 + \mathtt{size}(e) & \mathtt{size}(\mathtt{letrec}\ \{x_i = e_i\}_{i=1}^{n}\ \mathtt{in}\ e) = \mathtt{size}(e) + \sum \mathtt{size}(e_i) \\
\mathtt{size}(x\,\mathbf{m}-) & = 1 & \mathtt{size}(f\ e_1 \ldots e_n) = 1 + \sum \mathtt{size}(e_i) \\
\mathtt{size}(\nu x.P) & = 1 + \mathtt{size}(P) & \text{for constructors and operators } f
\end{array}
$$

Figure 2: Definition of `size` of expressions

imperative concurrent threads and industrial processes where the number of machines can be optimized since it is similar to required space.

The prerequisite is that the complete run of the processes is already given, insofar the optimization can be classified as offline. It can also be used in variants of job-shop-scheduling problems, where the number of machines has to be minimized and where the time is not relevant, e.g. [2].

The model is also extended to synchronization constraints in the form of a Boolean combination of conditions on simultaneous time points of two threads. The results for the space optimization for synchronization-free processes can be transferred to processes with synchronization and permits polynomial algorithms for fixed number of synchronization constructs (see Theorem 4.29) and therefore allows further analyses of space in more concrete scenarios.

The space-optimization of parallel processes can sometimes be also applied to the whole evaluation of a program in CHF. For example for processes that are deterministically parallel, there are no MVars and where the computation terminates.

The **structure of the paper** is first to introduce *CHF***GC* and a definition of a space improvement in section 2. and 3. A process-model, schedules and their space optimization is analyzed in section 4.

2 The Process Calculus *CHF*

In this section we give an informal presentation of the program calculus *CHF* which models a core language of Concurrent Haskell extended by futures, where more details can be found in [6, 7, 9]. Given a partitioned set of *data constructors c* such that each family represents a type T, then the data constructors of T are $c_{T,1}, \ldots, c_{T,|T|}$ where each $c_{T,i}$ has an arity $\mathrm{ar}(c_{T,i}) \geq 0$. For example, we assume that there is a type `Bool` with data constructors `True`, `False` and a type `List` with constructors `Nil` and `:` (written infix as in Haskell). The two-layered syntax of the calculus *CHF*, originally introduced in [6] has processes on the top-layer which may have expressions (the second layer) as subterms. Processes and expressions are defined by the grammars in Fig. 1 where *Var* is a countably-infinite set of variables, denoted with x.

Parallel processes are formed by parallel composition " $\mid$ ", a binary operator that is commutative and associative. ν-binders restrict the scope of variables and a concurrent thread $x \Leftarrow e$ evaluates the expression e and binds the result of the evaluation to the variable x. The variable x is also called the

future x. In a process there is (at most one) unique distinguished thread, called the *main thread* written as $x \overset{\mathsf{main}}{\Longleftarrow} e$. MVars are mutable variables which are empty or filled. If a thread wants to fill an already filled MVar $x\,\mathbf{m}\,e$ or empty an already empty MVar $x\,\mathbf{m}\,-$, then the thread blocks. The variable x is called the *name of the MVar*. Bindings $x = e$ are part of the global heap of shared expressions, where x is called a *binding variable*. A process is *well-formed*, if all introduced variables are pairwise distinct and there exists at most one main thread $x \overset{\mathsf{main}}{\Longleftarrow} e$.

Expressions *Expr* consist of a call-by-need lambda calculus and monadic expressions *MExpr* which model IO-operations. Functional expressions are built from variables, *abstractions* $\lambda x.e$, *applications* $(e_1\ e_2)$, *constructor applications* $(c\ e_1\ \ldots\ e_{\mathrm{ar}(c)})$, `letrec`*-expressions* $(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e)$, $\texttt{case}_T$*-expressions* for every type T, and `seq`*-expressions* $(\texttt{seq}\ e_1\ e_2)$. We abbreviate `case`-expressions as $\texttt{case}_T\ e\ \texttt{of}\ Alts$ where *Alts* are the `case`*-alternatives*. The `case`-alternatives must have exactly one alternative $(c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i)$ for every constructor $c_{T,i}$ of type T, where the variables $x_1, \ldots, x_{\mathrm{ar}(c_{T,i})}$ (occurring in the *pattern* $c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})}$) are pairwise distinct and become bound with scope e_i. In $(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e)$ the variables $x_1, \ldots, x_n$ are pairwise distinct and the bindings $x_i = e_i$ are recursive, i.e. the scope of x_i is $e_1, \ldots, e_n$ and e. We use $\{x_{g(i)} = s_{f(i)}\}_{i=j}^{m}$ for $x_{g(j)} = s_{f(j)}, \ldots, x_{g(m)} = s_{f(m)}$ and we abbreviate (parts of) `letrec`-environments as *Env* and thus e.g. write $\texttt{letrec}\ Env\ \texttt{in}\ e$.

Monadic operators `newMVar`, `takeMVar`, and `putMVar` are used to create and access MVars, the "bind"-operator `>>=` implements the sequential composition of IO-operations, the `future`-operator is used for thread creation, and the `return`-operator lifts expressions to monadic expressions.

Functional values are defined as abstractions and constructor applications. Moreover the monadic expressions $(\texttt{return}\ e)$, $(e_1\ \texttt{>>=}\ e_2)$, $(\texttt{future}\ e)$, $(\texttt{takeMVar}\ e)$, $(\texttt{newMVar}\ e)$, and $(\texttt{putMVar}\ e_1\ e_2)$ are called *monadic values*. A *value* is either a functional value or a monadic value. For typing we use a monomorphic type system and a small-step reduction relation is used for the execution of programs, where more information can be found in [7].

Informally, a standard reduction is an application of a reduction rule at a needed position, which is defined in [6, 7, 9], using $\mathbb{L}$-contexts: Every thread $x \Leftarrow e$ needs the evaluation of its expressions e, and a usual call-by-need strategy is used to find the corresponding redex. (Note that several threads may need the evaluation of the same redex.) *Successful processes* are the successful outcomes of the standard reduction (see [6, 7, 9]). They capture the behavior that termination of the main-thread implies termination of the whole program. A well-formed process P is *successful* and the standard reduction stops, if the main-thread returns `return` e where e is an expression.

We briefly recall the notion of contextual equivalence with may- and should-convergence as observations (see [6]). The concept is to equate processes P_1, P_2 whenever their observable behavior is indistinguishable if P_1 and P_2 are plugged into any process context. The process contexts $\mathbb{D} \in PCtxt$ are defined using the grammar $\mathbb{D} := [\cdot] \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}$. As observations we use may- and should-convergence:

Definition 2.1. *A process P* may-converges *(written as $P{\downarrow}P'$), iff it is well-formed and reduces to a successful process P'. If we do not need P' then we may write $P{\downarrow}$. If $P{\downarrow}$ does not hold, then P* must-diverges *written as $P{\Uparrow}$.*

A process P should-converges *(written as $P{\Downarrow}$), iff it is well-formed and remains may-convergent after reductions. If P is not should-convergent then we say P* may-diverges *written as $P{\uparrow}$.*

Definition 2.2. Contextual approximation $\leq_c$ *is defined as* $\leq_c := \leq_\downarrow \cap \leq_\Downarrow$, contextual may-equivalence $\sim_{\downarrow,c}$ *is defined as* $\sim_{\downarrow,c} := \leq_\downarrow \cap \geq_\downarrow$, *and* contextual equivalence $\sim_c$ *on processes is defined as* $\sim_c := \leq_c \cap \geq_c$ *where for* $\xi \in \{\downarrow, \Downarrow\}$*:* $P_1 \leq_\xi P_2$ *iff* $\forall \mathbb{D} \in PCtxt : \mathbb{D}[P_1]\xi \Rightarrow \mathbb{D}[P_2]\xi$.

A *program transformation* γ on processes is a binary relation on processes. It is *correct* iff $\gamma \subseteq \sim_c$.

Remark 2.3. *We added an empty process $\emptyset$ which is not in the syntax of CHF (see [6, 7]). This addition is without problems, since $\emptyset\Uparrow$ and thus it is contextually equivalent to any other process that must-diverges.*

2.1 Garbage Collection

The calculus *CHF* (see [6, 7]) and the variant CHF^* in [9] that has a slightly modified operational semantics for the case-construct both omit an explicit treatment of garbage collection. Ignoring garbage collection is not an option, if we want to analyze space improvements, hence we add a formalism for eager garbage collection analogous to the one in [8]. We explicity add a rule to CHF^* obtaining a calculus CHF^*GC. We assume that processes are in ν-prefix form and the variable convention holds before (gc) is applied.

Definition 2.4. *CHF^*GC is the calculus CHF^* with the additional rule (gc): $P_1 \xrightarrow{gc} P_2$, where P_2 is generated from P_1, such that a maximal set M of the following components and bindings is removed: $x = e$, $x \Leftarrow e$ if this is not the main thread, $x\,\mathbf{m}\,e$, $x\,\mathbf{m}\,-$, and letrec-bindings $x = e$, such that the following conditions hold:*

1. *The variables x in top-level sharing, threads, and MVars are ν-bound or let-bound.*
2. *The variables x do not occur as ν-bound or free variables in P_2.*
3. *There is no variable y that occurs in P_2 and which is also free in M.*

After this operation also empty letrec-environments and νx-binding operators that do not bind any x are removed. The standard reduction is modified, such that (gc) has priority, whenever it is possible to apply it and if something is removed. We say a process is garbage-free *iff (gc) does not change the process.*

It is not hard to see that two consecutive (gc)-standard reductions are not possible. It is also not hard to see that (gc) commutes with other standard reductions and hence that may-and must-convergence are not modified after an application of (gc). Hence CHF^* and CHF^*GC are equivalent.

3 Space Improvements in CHF^*

As space measure we use a generalization of the space measure of [8], that does not count variables to achieve compatibility with abstract machines. Since we want to build a theory about space and not garbage, we use CHF^*GC and interpret the space usage of reduction rules (r) from CHF^* as follows. Let P_1 be a garbage-free process and $P_1 \xrightarrow{r} P_2$ or $P_1 \xrightarrow{r} P_2' \xrightarrow{gc} P_2$ be a standard reduction sequence where P_2 is garbage-free. Then the space effect of $\xrightarrow{r}$ is only compared on P_1, P_2 when these are garbage-free.

Definition 3.1. *The size $\mathtt{size}(e)$ of an expression e and the size of a process P are defined in Fig. 2.*

Definition 3.2. *The space measure $spmax(Red)$ of the successful standard reduction Red of a process P is the maximum of all sizes $\mathtt{size}(P_i)$ during the whole standard reduction sequence, Red where only sizes of the garbage-free processes are counted. The space measure of a process P is defined as $spmax(P) = min\{spmax(Red) \mid Red \text{ is a successful standard reduction of } P\}$.*

The reason for not counting the sizes directly before a garbage collection is that the calculus and abstract machines may create bindings that may be garbage and would be immediately garbage collected after the reduction step. Taking this garbage into account would distort the reasoning about measurement in particular if these bindings have a large size (more information about this can be found in [8]). This principle of measuring space in a small-step calculus is also used in [4].

Definition 3.3. *Let P_1 and P_2 be two well-formed processes such that one of the following holds.*

1. *$P_1\!\uparrow$ and $P_2\!\uparrow$, or*
2. *$P_1\!\downarrow, P_2\!\downarrow$, $P_1 \sim_c P_2$; and $\forall \mathbb{D} \in PCtxt : spmax(\mathbb{D}[P_1]) \leq spmax(\mathbb{D}[P_2])$*

Then P_1 space-improves *P_2, written $P_1 \leq_{spmax} P_2$. A program transformation $\xrightarrow{PT}$* is a space-improvement *if for all processes P_1, P_2: $P_1 \xrightarrow{PT} P_2$ implies that P_2 space-improves P_1*

Proposition 3.4. *An example for a transformation that is a space-improvement is the following rule, if executed as standard reduction ($\mathbb{L}$ are contexts that define the positions where a standard reduction can be applied):* (lbeta) : $\mathbb{L}[((\lambda x.e_1)\ e_2)] \to \nu x.(\mathbb{L}[e_1] \mid x = e_2)$,

4 Schedules

In the following we analyze the local space requirement of processes by considering an abstract model. The assumptions underlying the abstraction is that the processes use shared memory for their local data structures, that they may pause, start or stop at certain times, and that synchronization and communication are actions that occur at certain time points between two processes.
Every process is abstractly modeled by its space requirements, given as a list of integers. In addition we later add constraints expressing simultaneous occurrence of time points of different processes as well as start-points and end-points of processes.

4.1 Schedules for Space

Definition 4.1. *A (space-)process is a non-empty list of non-negative integers.*
Let $p_1, \ldots, p_n$ be n processes. Then an index-schedule *is a list of n-tuples, starting with $(1, \ldots, 1)$, ending with $(m_1, \ldots, m_n)$, where m_i is the length of list p_i for $i = 1, \ldots, n$, and if $(a_1, \ldots, a_n)$, $(b_1, \ldots, b_n)$ are two consecutive tuples in the index-schedule, then for all i: $a_i = b_i$, or $a_i + 1 = b_i$.*

An index-schedule I generates a usual schedule *of n space-processes, which is a list of n-tuples $\{(p_1(a_1), \ldots, p_n(a_n)) \mid (a_1, \ldots, a_n)$ is a tuple from $I\}$.*

Definition 4.2. *The space usage $sp(S)$ of a schedule S is the maximum of the sums of the elements in the tuples in S, i.e. $sp(S) = \max\{\sum_{i=1}^{m} a_i \mid (a_1, \ldots, a_m) \in S\}$. The maximal necessary space $spmax(p_1, \ldots, p_n)$ for n processes $p_1, \ldots, p_n$ is the minimum of the space usages of all schedules of $p_1, \ldots, p_n$, i.e.* $\min\{sp(S)) \mid$ *S is a schedule for $p_1, \ldots, p_n\}$.*

Example 4.3. *For two processes $[1,7,3], [2,10,4]$ the value $spmax(.)$ is 11, by first running the second one and then running the first. I.e. such a space-optimal schedule is* $[(1,2), (1,10), (1,4), (7,4), (3,4)]$.

4.2 Reduction of Schedules of Two Processes

We first consider optimization of the space of two processes since arguments are easier to grasp. The idea is to first compute a standard form of processes and to define the optimization only on the standard forms. First we define the standardization of processes using patterns. We argue that certain patterns in processes permit to compute *spmax* from smaller processes.

Definition 4.4. *The trivial pattern M_0 is $a_i = a_{i+1}$. There are two nontrivial patterns: The first pattern M_1 is $a_i \leq a_{i+1} \leq a_{i+2}$ and the second pattern M_2 is $a_i \geq a_{i+1} \geq a_{i+2}$.*
A pattern matches a process *at index i, if for index i the conditions are satisfied.*
If pattern M_0 matches a process, then a_{i+1} can be removed from the process. If the pattern M_1 or M_2 match a process, then a_{i+1} can be removed.

Proposition 4.5. *Let P_1, P_2 be two processes. If pattern M_0 matches one of the processes, then let P_1', P_2' be the processes after removal of subsequent equal space entries. Then $spmax(P_1, P_2) = spmax(P_1', P_2')$.*

In the following we use $(a:l)$ for adding a at the front of list l and l_1++l_2 for appending two lists.

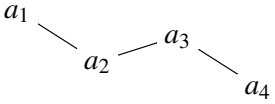
Proposition 4.6. *Let P_1, P_2 be two processes. If pattern M_1 or pattern M_2 matches one of the two processes, then let P_1', P_2' be the processes P_1, P_2 modified as follows: If there are three consecutive numbers a_1, a_2, a_3 in P_1 that match M_1 or M_2, then remove a_2 from P_1. The same for P_2. Then $spmax(P_1, P_2) = spmax(P_1', P_2')$.*

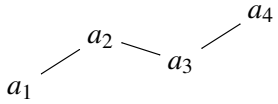
By exhaustive application we can assume that the pattern M_0,M_1 and M_2 above do not occur in processes which means that the processes are like a zig-zag:

Definition 4.7. *If in a process P every strict increase is followed by a strict decrease and every strict decrease is followed by a strict increase, then the process P is called a* zig-zag process.

Now we show that there are more complex patterns that can also be used to reduce the processes before computing *spmax*.

Definition 4.8. *The following patterns M_3, M_4 are like stepping upstairs and downstairs, respectively. M_3 consists of a_1, a_2, a_3, a_4, with $a_1 > a_2 < a_3 > a_4$ and $a_1 \geq a_3, a_2 \geq a_4$. M_4 consists of a_1, a_2, a_3, a_4, with $a_1 < a_2 > a_3 < a_4$ and $a_1 \leq a_3, a_2 \leq a_4$. If M_3 or M_4 matches then eliminate a_2, a_3.*

M_3 : 

M_4 : 

We show that the patterns can be used to restrict the search to special space-processes:

Lemma 4.9. *Let P_1, P_2 be two space-processes, where patterns M_0, M_1, M_2 are not applicable. If one of the patterns M_3, M_4 matches one of the processes, then it is sufficient to check the shortened P_1', P_2' for the space-minimum.*

We show that it is sufficient to analyze processes where the first and last elements are valleys.

4.3 Standard Form of Processes

If the goal is to compute the optimal space, then there are several reduction operations on processes that ease the computation and concentrate on the hard case. First we show that one-element processes can be excluded, second that processes with start- or end-valleys can be reduced by omitting elements. Then we show that through the use of 5 patterns $M_0 \ldots, M_4$ for reductions we can concentrate on special forms of zig-zag-processes, so-called midzz.

Proposition 4.10. *If $P_1 = [a_1]$ and $P_2 = [b_1, \ldots, b_n]$, then $spmax(P_1, P_2) = a_1 + spmax(P_2)$.*

Proposition 4.11. *Let $P_1 = [a_1, \ldots, a_n]$, $P_2 = [b_1 \ldots, b_m]$ be two processes. If a_1 is not a valley, then let $P_1' = [a_2, \ldots, a_n]$. Then for $m' = spmax(P_1', P_2)$, we obtain $spmax(P_1, P_2) = max(a_1 + b_1, m')$.*
The same holds (symmetrically) if P_1 does not end with a valley.

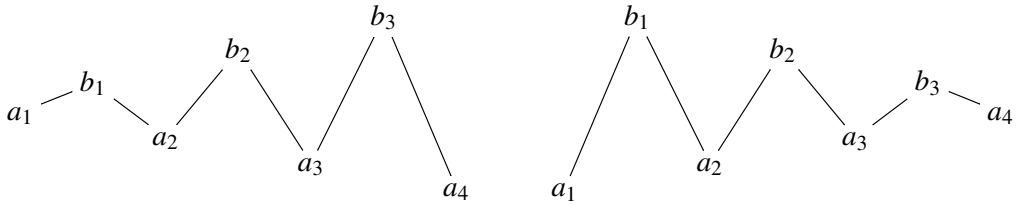
Lemma 4.12. *We can assume that processes P_1, P_2 are both of length at least 3 for computing the optimal space.*

Definition 4.13. *A process $[a_1, b_1, \ldots, a_n, b_n]$ or $[a_1, b_1, \ldots, a_n]$ is a monotone-decreasing zig-zag (mdzz), iff $a_i < b_j$ for all i, j, and $a_1, a_2, \ldots$ is monotone increasing, and $b_1, b_2, \ldots$ is monotone decreasing.*

A process $[a_1, b_1, \ldots, a_n, b_n]$ or $[a_1, b_1, \ldots, a_n]$ is a monotone-increasing zig-zag (mizz), iff $a_i < b_j$ for all i, j, and $a_1, a_2, \ldots$ is monotone decreasing, and $b_1, b_2, \ldots$ is monotone increasing.

A process is midzz*, if it is an mizz followed by an mdzz.*

Typical graphical representations of an mizz and mdzz are:



Proposition 4.14. *A process such that none of the patterns M_0, M_1, M_2, M_3, M_4 matches, is a midzz.*

Note that the definition of midzz permits the simplified case that the process is a mizz or mdzz.

Lemma 4.15. *Let P be a process that starts and ends with peaks. Then the application of the patterns $M_0, \ldots, M_4$ with subsequent reduction always produces a process that also starts and ends with peaks.*

Lemma 4.16. *Let P be a midzz-process, where no pattern M_0, M_1, M_2, M_3, M_4 applies and which is of length at least 3 and does not start nor end with a local peak: Then a midzz-process has one or two global peaks, it has one or two global valleys, but not two global peaks and two global valleys at the same time.*

We show that the optimal space for 2 midzz schedules can be computed in polynomial time.

Algorithm 4.17. Algorithm for Left-Scan *We describe an algorithm for 2 processes that makes a left-scan until a valley is reached. This contributes to the computation of the optimal space of 2 midzz processes P_1, P_2. In $P_i, i = 1, 2$, the following indices are fixed:*

> *$I_{ends,i}$, which is the index in P_i of the global valley, if it is unique and the rightmost valley if there are two minimal valleys.*

Note that these two indices are neighbors for every i and for this algorithm we assume that both processes P_1, P_2 are non-exceptional: They have at least three elements each and there are at least the peak and the two valleys left and right of the peaks. The algorithm starts from left and proceeds to the right. Let a_j be the list for P_1 and b_j be the process-list of P_2.

1. *It starts at the left and initializes Max with $a_1 + b_1$. There are two indices I_1, I_2, which indicate the current valley-positions in P_1, P_2, respectively during the run. Initially, $I_1 = 1, I_2 = 1$.*
2. *The step of the algorithm which is applied iteratively is as follows:*
   (a) *If $a_{I_1+1} + b_{I_2} \leq Max$ and $I_1 + 2 \leq I_{ends,1}$ then step forward: I_1 is set to $I_1 + 2$. Goto* (2).
   (b) *If $a_{I_1} + b_{I_2+1} \leq Max$ and $I_2 + 2 \leq I_{ends,2}$, then step forward: I_2 is set to $I_2 + 2$. Goto* (2)
3. *If none of the previous steps is possible, then there are several cases:*
   (a) *$I_1 \neq I_{ends,1}$ and $I_2 \neq I_{ends,2}$, then Max has to be strictly increased: The new Max is $Max := min\{a_{I_1+1} + b_{I_2}, a_{I_1} + b_{I_2+1}\}$. Due to the conditions, this new value is strictly greater than the previous Max. Goto* (2).

(b) $I_1 = I_{ends,1}$ *and* $I_2 \neq I_{ends,2}$*, then strictly increase Max:* $Max = a_{I_1} + b_{I_2+1}$*. Goto* (2).
(c) $I_1 \neq I_{ends,1}$ *and* $I_2 = I_{ends,2}$*, then strictly increase Max:* $Max = a_{I_1+1} + b_{I_2}$*. Goto* (2).
(d) If $I_1 = I_{ends,1}$ *and* $I_2 = I_{ends,2}$*, then the stop condition is reached and Max is returned.*

The right-to-left algorithm is the symmetric version and yields also a maximum value for the right part.

Algorithm 4.18. Computation of *spmax* **for two processes** *First compute* Max_{left} *and* Max_{right} *using the left-to-right scan and right-to-left scan and finally compute the maximum of* Max_{left} *and* Max_{right}*.*

Remark 4.19. *Note that the maximum usage of space may occur in indices that do not correspond to the peaks. Intuitively, this happens, if the valleys close to the peaks are very deep.*

Theorem 4.20. *For two midzzs* p_1, p_2*, a space-optimal schedule can be constructed in polynomial time. In addition the maximally necessary space can be computed in polynomial time.*

Remark 4.21. *The algorithm is presumably quadratic due to the pattern match. The other steps only contribute a linear component to the complexity.*

4.4 Calculation for Space Processes

Our interpreter CHFi calculates all possible reduction paths that may be caused by the nondeterminism introduced by access-conflicts on MVars together with the complete space-profile for each path, which may be helpful in analyzing and understanding space-behavior of CHF-programs. The program can be downloaded here: `www.ki.cs.uni-frankfurt.de/research/chfi` .

Let us assume that we apply this only in the case that processes use disjoint sets of MVars, then their use is deterministic and can be modeled without MVars. Hence our interpreter calculates only a single reduction path per process. This enables us to immediately have space processes which can be optimized using the algorithm that computes the optimal space. Now CHFi and our algorithm can be used together in the following cases, where P_1 and P_2 are space processes that can be directly calculated using the CHFi using the given corresponding CHF-programs:

1. Since CHFi computes the possible parallel executions using an eager strategy, not every possible parallel evaluation is considered in the interpreter. Using the single processes and the space optimization algorithm, it is possible to compare the optimum with the (probably larger) CHFi-computed space usage and so we propose to extend CHFi by an implementation of the space optimization algorithm.
2. Let t be a transformation that is applied to processes P_1 and P_2, yielding P_1' and P_2'. Let S be the result of using the algorithm with P_1 and P_2 and S' with P_1' and P_2' respectively.
If $spmax(P_1,P_2) < spmax(P_1',P_2')$ then t is not a space improvement. If $spmax(P_1,P_2) \geq spmax(P_1',P_2')$ then t is a space improvement under conditions on the use of P_1, P_2 in the concurrent program.

4.5 Many Independent Processes

Let us assume in this section that there are n processes $P_1, \ldots, P_n$ for $n \geq 2$. We generalize the reduction method for processes from 2 to n processes.

Proposition 4.22. *For n processes* $P_1, \ldots, P_n$*, the patterns* $M_0, M_1 \ldots, M_4$ *can be applied to reduce processes without changing the minimal space for scheduling the processes.*

Proposition 4.23. *For n processes* $P_1, \ldots, P_n$*, we can assume that the processes do not start or end with local peaks.*

We describe an algorithm to compute the optimal space for a schedule of n midzz processes $P_1, \ldots, P_n$, that do not have peaks at the start or at the end and have at least 3 elements. For $i = 1, \ldots, n$ there is a unique peak $peak_i$ of P_i, which is of length at least 3. There are also two unique valleys: $valley_{i,left}$ and $valley_{i,right}$. These are the left and right neighbors of the corresponding peaks. Since we assumed that the processes P_i are non-exceptional, every process has at least three elements each and there are at least the peak and the two valleys left and right of the peak.

Algorithm 4.24. Algorithm for Left-Scan of n processes *We describe an algorithm for $n \geq 2$ processes that makes a left-scan until a valley is reached. This contributes to the computation of the optimal space of n midzz processes $P_1, \ldots, P_n$. The following indices in P_i are fixed:*

> *$I_{i,ends}$, which is the index in P_i of the global valley, if it is unique and of the rightmost global valley if there are two minimal valleys. global peak if there are two maximal peaks.*

Note that for every i, these two indices are neighbors and for this algorithm we assume that all processes P_i are non-exceptional: They are midzzs, have at least three elements each and they do not start or end with a local peak.

1. *It starts at the left and initializes Max with $\sum_i p_{i,1}$ There are indices I_i for $i = 1, \ldots, m$, which indicate the current valley-positions in P_1, P_2, respectively, during the run. Initially, $I_i = 1$ for all i.*
2. *The step of the algorithm which is applied iteratively is as follows:*
   (Nondeterministically) select some $j \in \{1, \ldots, n\}$ with $p_{j,I_j+1} + \sum_{k \neq j} p_{k,I_k} \leq Max$ and $I_j + 2 \leq I_{j,ends}$. If such an index j is selected, then the algorithm steps forward: I_j is set to $I_j + 2$ and proceeds at (2)*. If there is no such j then go on to the next item.*
3. *If for all i, $I_i = I_{ends,i}$, then the algorithm stops and returns the current Max.*
4. *We have to increase Max: For $k = 1, ..n$, if $I_k = I_{ends,k}$, then $M_k := \infty$. If $I_k < I_{ends,k}$ then let $M_k := p_{I_k+1} + \sum_{h \neq k} p_{h,I_h}$. Let m be one of the indices such that M_m is minimal among all M_k. Then $Max := M_m$ and $I_m := I_m + 2$. The other indices I_k are unchanged. Goto* (2)*.*

The right-to-left algorithm is the symmetric version and yields also a maximum value for the right part.

Algorithm 4.25. Computation of *spmax* **for n processes** *First compute M_{left} and M_{right} using the left-to-right and right-to-left scan and finally compute the maximum of M_{left} and Max_{right}.*

A process in standard form is a midzz of length at least 3 and does not start or end with a local peak.

Theorem 4.26. *Algorithm 4.25 computes the optimal space for schedules of n processes in standard form.*

Theorem 4.27. *If there are N processes $P_1, \ldots, P_N$, then the optimal space and an optimal schedule can be computed in (asymptotic) time polynomial in n, where n is the size of input.*

4.6 Processes with Synchronizations

Definition 4.28. *There may be various forms of synchronization restrictions. We will only use those forms of fundamental restrictions:*

1. *$simul(P_1, P_2, i_1, i_2)$ for process P_1, P_2 there are two indices i_1, i_2 that must happen simultaneously.*
2. *$starts(P_1, P_2, i)$ process P_1 starts at index i of process P_2.*
3. *$ends(P_1, P_2, i)$ process P_1 end at index i of process P_2.*

For a set R of restrictions only schedules are permitted that obey all restrictions. This set is also called a set of deterministic *restrictions. We also permit Boolean formulas over such basic restrictions. In this case the permitted schedules must obey the formula.*

We show that for n processes and a Boolean restriction there is an algorithm for computing the optimal space and an optimal schedule that has a exponential complexity, where the exponent is $b \cdot n$ where b is the number of basic restrictions and N is the number of processes.

Theorem 4.29. *Let there be N processes and a set B of boolean restrictions where b is the number of basic restrictions in B and the size of the input is n. Then there is an algorithm to compute the optimal space and schedule of worst case asymptotic complexity of $O(p(n) \cdot n^{O(b \cdot N)})$, where p is a polynomial.*

Corollary 4.30. *Let there be N processes and a set B of Boolean restrictions where b is the number of basic restrictions in B and the size of the input is n. Assume that the number N of processes and the size of B is fixed. Then there is a polynomial algorithm to compute the optimal space and schedule.*

Theorem 4.31. *In the general case of synchronization restrictions, the problem of finding the minimal space is NP-hard.*

4.7 Limitations and Extensions

Our space optimization algorithm in its current form is not applicable to schedules for CHF-threads that use shared data structures. One problem is that the common use of data structures cannot be controlled in a call-by-need evaluation, in particular since the optimization algorithm rearranges the execution sequences. An extension to shared data is future work.

If all restrictions are deterministic, then it is currently open whether the optimization algorithm can be adapted or extended to a polynomial time algorithm. In this case the input schedules are like a directed graph with space-labels. This prevents to use the simple schedule-modifications for standardizing the schedules and reducing the problem to zig-zag schedules.

If there are only deterministic restrictions, then there may a polynomial algorithm for optimizing space in special cases, for example, the case where all schedules are synchronized in a fixed number of time points, and where the number of time points is not restricted. The idea is to apply the optimization algorithm in the sequence of time intervals.

The addition of deterministic restrictions also leads to the question whether the set of all restrictions is impossible or not. For example, the definition of deterministic restrictions also permits sets of such restrictions that are incompatible with a linear order of time.

Restrictions as Boolean formulas are perhaps too general, since the problem of finding a space optimal schedule in this case is NP-hard, as shown above. The particular question whether such a Boolean formula of restrictions permits at least one schedule appears also to be NP-hard.

5 Conclusion and Future Research

We developed a polynomial offline-algorithm that optimizes a given set of processes w.r.t. space that is applicable to concurrent lazy-evaluating languages and others because of the generality of the input. We showed that with a fixed set of synchronization-requirements the algorithm remains polynomial and the general case is NP-complete. Future work is the study of additional examples of synchronization-restrictions and the extension of the theory to prove space improvements in all contexts.

References

[1] Haskell Community (2016): *Haskell, an advanced, purely functional programming language*. Available at `www.haskell.org`.

[2] M. R. Garey & D. S. Johnson (1977): *Two-Processor Scheduling with Start-Times and Deadlines*. *Siam J. Comput.* 6(3), pp. 316–426.

[3] Jörgen Gustavsson & David Sands (1999): *A Foundation for Space-Safe Transformations of Call-by-Need Programs*. *Electr. Notes Theor. Comput. Sci.* 26, pp. 69–86, doi:10.1016/S1571-0661(05)80284-1.

[4] Jörgen Gustavsson & David Sands (2001): *Possibilities and Limitations of Call-by-Need Space Improvement*. pp. 265–276, doi:10.1145/507635.507667.

[5] S. Peyton Jones, A. Gordon & S. Finne (1996): *Concurrent Haskell*. In Guy L. Steele Jr. H.-J. Boehm, editor: *Proc. 23th ACM Principles of Programming Languages*, ACM, pp. 295–308.

[6] David Sabel & Manfred Schmidt-Schauß (2011): *A contextual semantics for concurrent Haskell with futures*. In Peter Schneider-Kamp & Michael Hanus, editors: *Proc. 13th ACM PPDP 2011*, ACM, pp. 101–112.

[7] David Sabel & Manfred Schmidt-Schauß (2011): *A Contextual Semantics for Concurrent Haskell with Futures*. Frank report 44, Institut für Informatik. Goethe-Universität Frankfurt am Main. `http://www.ki.informatik.uni-frankfurt.de/papers/frank/`.

[8] Manfred Schmidt-Schauß & Nils Dallmeyer (2018): *Space Improvements and Equivalences in a Functional Core Language*. In Horatiu Cirstea & David Sabel, editors: Proceedings Fourth International Workshop on *Rewriting Techniques for Program Transformations and Evaluation*, Oxford, UK, 8th September 2017, *Electronic Proceedings in Theoretical Computer Science* 265, Open Publishing Association, pp. 98–112, doi:10.4204/EPTCS.265.8.

[9] Manfred Schmidt-Schauß, David Sabel & Nils Dallmeyer (2017): *Improvements for Concurrent Haskell with Futures*. Frank report 58, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main.