Improvements for Concurrent Haskell with Futures*

Manfred Schmidt-Schauß, David Sabel, and Nils Dallmeyer

Goethe-University, Frankfurt, Germany
{schauss,sabel,dallmeyer}@ki.cs.uni-frankfurt.de

Technical Report Frank-58

Research group for Artificial Intelligence and Software Technology
Institut für Informatik,
Fachbereich Informatik und Mathematik,
Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany

Abstract. We propose a model for measuring the runtime of concurrent programs by the minimal number of evaluation steps. The focus of this paper are improvements, which are program transformations that improve this number in every context, where we distinguish between sequential and parallel improvements, for one or more processors, respectively. We apply the methods to CHF, a model of Concurrent Haskell extended by futures. The language CHF is a typed higher-order functional language with concurrent threads, monadic IO and MVars as synchronizing variables. We show that all deterministic reduction rules and 15 further program transformations are sequential and parallel improvements. We also show that introduction of deterministic parallelism is a parallel improvement, and its inverse a sequential improvement, provided it is applicable. This is a step towards more automated precomputation of concurrent programs during compile time, which is also formally proven to be correctly optimizing.

1 Introduction

Motivation and Goals. A current trend in programming and programming languages is towards distributing and parallelizing computation tasks. The design and implementation of algorithms for distributed and parallelized computing is a complex engineering task and optimizing such algorithms and systems is an art. A natural feature are the different and unpredictable speeds of the various sub-computations, and the need for controlling and synchronizing them. It is

* The first and third authors are supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SCHM 986/11-1, and the second author is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA2908/3-1.

known that the functional programming paradigm can contribute to this area, in particular lazy functional programming in providing the tools for specification of computation results, data dependencies and data flow, without exactly specifying the exact sequence of evaluation steps [14, 8, 24, 4, 17].

A method to optimize concurrent programs is to apply source-to-source program transformations. Concerning so-called deterministic parallelism they may even introduce concurrency and parallelism into sequential programs and thus "parallelize" the programs [3, 39, 18]. An indispensable requirement is that the applied transformations do not change the meaning of the program, i.e. that they are correct and thus leave the semantics of the programs unchanged. The next crucial question is how they influence the resource behavior of the programs. Ideally, those transformations are preferable, which optimize the programs w.r.t. time, space and/or the number of required processors. In sequential functional languages the so-called improvement theory was developed by [20], recently revived e.g. by [11, 33, 34], to provide a strong notion of when a program transformation optimizes a program in any case. A study of improvements in a nondeterministic setting is in [16], who study a call-by-name lambda-calculus with McCarthy's amb-operator, and define and analyze a cost simulation and cost equivalence for may- and must-convergence. However, to the best of our knowledge, for a concurrent programming language with shared memory and side-effecting computations there is no corresponding analysis. It is also unclear how to transfer the simulation-techniques of [16] into such a program calculus.

We consider an abstract model of concurrent processes to illustrate the notion of improvements in a concurrent setting. Let $\mathcal{P}$ be a set of programs, and for $P \in \mathcal{P}$, let $rl(P) \subseteq \mathbb{N}$ be the set of possible lengths of successful reduction sequences of P. For example, a concurrent (nondeterministic) program P_0 that reduces in 3, 5, and 100 reduction steps to a value has $rl(P_0) = \{3, 5, 100\}$, and a program P' without any successfully terminating reduction sequence has $rl(P') = \emptyset$. Let $\mathcal{C} \subseteq (\mathcal{P} \to \mathcal{P})$ be the set of contexts. We assume that $\sim$ as program equivalence is already given as a congruence, i.e. $P_1 \sim P_2 \implies C(P_1) \sim C(P_2)$. We say program P_1 *improves* P_2 (w.r.t. the runtime), notation $P_1 \preceq P_2$, if $P_1 \sim P_2$, and $\forall C \in \mathcal{C} : \min(rl(C[P_1])) \leq \min(rl(C[P_2]))$ where $\min(\emptyset) = \infty$. This is consistent with the same notion for deterministic programs where $|rl(P)| \leq 1$. It is the same as requiring that for every successful reduction sequence of $C[P_2]$ of length n, there is a successful reduction sequence of $C[P_1]$ of length at most n. Hence it is also consistent with the may-convergence part of the definition in [16].

We give arguments in favor of this definition. In a realistic concurrent language, this notion does not only mean to minimize the reduction length, but due to the in-all-context condition it is finer: it compares the minimal reduction length for every resulting value. The reason is that the improvement notion is contextual, and for every (finite) data value v a context C_v can be programmed that accepts the value and otherwise loops. Our notion focuses on shortest reduction sequences and thus it immediately prefers reduction sequences without redundant reduction steps, for example of sub-processes that do not contribute to the final value. Also, other alternatives to using the minimum are question-

able, for example, taking the supremum may result in ∞ for simple recursive non-deterministic programs like $p = 1 \oplus p$, (where $\oplus$ means nondeterministic choice). That p is a proper improvement of $p' = (1 \oplus 1) \oplus p'$ can only be justified by the minimum-based definition.

Since the matter of concurrency is complex, we restrict the focus of the paper:

- We only consider improvements of the runtime and thus ignore any considerations of space and other resources.
- We restrict the runtime-observation to the minimum-based definition, which can be seen as optimizing programs for the best-case, i.e. for a perfect scheduler which always chooses the shortest evaluation.
- Concerning the reduction length of successful evaluations we will consider two definitions which are both suitable for the concurrent setting: A single-processor model where a reduction is a sequence of interleaved steps which stem from the concurrent processes, and a multi-processor model where parallel reduction steps are allowed such that different concurrent processes make progress at the same time.
- We will work with a specific concurrent (and lazy functional) programming language whose semantics is well-analyzed, such that we can reuse existing results (and techniques) on the correctness of transformations.

Thus, there are two goals: we want to develop an improvement theory for concurrent programming and we want to analyze concrete transformations with regards to being improvements. For accomplishing the second goal, we will consider CHF as an expressive concurrent language model.

Focusing on the Program Calculus CHF.* Concurrent Haskell was proposed in [24], and implemented in the Glasgow Haskell Compiler [23, 25]. There is an *imperative level* (the action layer) which sequentializes computation by Haskell's monadic programming features (see e.g. [26, 40, 23]) and it permits the execution of side-effects like starting further threads and modifying external storage. The pure functional level is the core part. The combination of monadic and pure functional programming is a compromise between the need for sequential actions and the unspecified sequence of evaluating pure functional expressions.

As for deterministic (lazy) functional programming, concurrent (and lazy) functional programming leaves the sequence unspecified in its pure part, hence permits lots of different possible parallel and distributed evaluations for the same initial situation [24, 4, 22, 21]. Optimizing a concurrent functional program by program transformations puts several issues. The first issue is whether the program modifications are correct, which depends on the chosen semantics; the second issue is the notion of optimization, which depends on the chosen model of resources and their usage. The difficulties with concurrency are highlighted by the fact that two nontrivial and sensible programs P and P' may be equivalent w.r.t. the chosen semantics, and both may have an infinite number of different evaluations leading to different values.

We want to model this in a mathematical clean way that is also applicable to practical applications and has a potential to be applied to Concurrent Haskell.

```
data Tree  = Node Tree Tree | Leaf N
f :: A → A → A                g :: N → A                someTree :: Tree
```

```
mainPure =
  let res = (calcPure someTree)
  in  seq res (return res)
calcPure (Leaf n)   = g n
calcPure (Node l r) =
  (calcPure l) `f` (calcPure r)
```

```
mainFut = calcFut someTree
calcFut (Leaf n) =
 let res = (g n)
 in seq res (return res)
calcFut (Node l r) = do
  lres <- future (calcFut l)
  rres <- future (calcFut r)
  let res  = (lres `f` rres)
  seq res (return res)
```

```
mainMVar = do
   up  <- newEmptyMVar
   calcMVar someTree up
   val <- takeMVar up
   return val
calcMVar (Node l r) up = do
   leftTreeVal <- newEmptyMVar
   rightTreeVal <- newEmptyMVar
   forkIO (calcMVar l leftTreeVal)
   forkIO (calcMVar r rightTreeVal)
   v1 <- takeMVar leftTreeVal
   v2 <- takeMVar rightTreeVal
   let w = (v1 `f` v2)
   seq w (putMVar up w)
calcMVar (Leaf n) up =
   putMVar up (g n)
```

Fig. 1. A pure and two concurrent implementations of binary tree-fold

As *concurrent language model* we employ *CHF** (Concurrent Haskell with Futures), which is a semantically equivalent variant of CHF [28, 29]. It captures the semantics of a variant of Concurrent Haskell extended by so-called futures [7, 22, 21, 18] which allow to declaratively use the result of concurrent computations. Our futures are related to the IVars of [18] who use a technique similar to futures in their deterministic parallel functional language. CHF borrows techniques from the call-by-need calculus [2] and from the pi-calculus [19]. Our choice of the example calculus is appropriate, since it is well investigated w.r.t. methods and results on the correctness of transformations. *CHF** is a process calculus which comprises shared memory in the form of Concurrent Haskell's MVars, named threads (i.e. so-called futures) and heap bindings: a finished thread $y \Leftarrow e$ is turned into a global binding $y = e'$, where e' is the value of e. On the expression level there are monadic IO-computations and pure functional expressions. The latter extend the lambda calculus by data constructors, case-expressions, recursive let-expressions, and Haskell's `seq`-operator for sequential evaluation. *CHF** comes with a monomorphic type system with recursive types where polymorphic data constructors are monomorphically instantiated. A small-step operational semantics tells us which (sequential and parallel) reduction sequences are possible [28]. A reduction sequence is successfully terminated, when the main thread has successfully finished its computation.

The *semantics* of *CHF** is a contextual semantics that compares processes in all possible contexts by may-convergence and should-convergence as proposed by [28], where the latter is different from must-convergence. This notion of semantic equality of programs is our criterion of correctness of a transformation. It is an extension of the contextual semantics of pure functional expressions to

nondeterministic processes. It is able to distinguish a program P where all reduction sequences are successful from a program P' with the same outcome and one additional reduction sequence that gets stuck (for example by a deadlock). A type system (a weak one like a monomorphic one is sufficient) is indispensable, since without types the contextual semantics distinguishes too many expressions and thus is of restricted use (for example `(map id)` and `id` are different without types). [28, 29] present different techniques for recognizing semantic equality, and also some results on correctness of transformations.

Improvements in CHF.* We work with two *resource models* for the runtime, the length of a sequential interleaved reduction sequence (the work done), and the length of a parallel reduction sequence, which represents the runtime for a fixed number of processors. If a program P is given, we look for the *minimum* of the lengths of all interleaved reduction sequences (that successfully stop) from P. Our criterion for a *CHF**-program P being an improvement of another *CHF**-program P' is that in all process-contexts $\mathbb{D}$, the minimal reduction length of $\mathbb{D}[P]$ is not greater than the minimal reduction length of $\mathbb{D}[P']$. We will argue in this paper that this makes perfect sense and support this by an analysis of the improvement relation. For the parallel resource model, we compare the minimal lengths of the parallel reduction sequences before and after a transformation.

An example for a particular transformation is $(\texttt{return}\ e_1) \gg\!= e_2 \to (e_2\ e_1)$ where `>>=` is the bind-operator for a monadic combination of two actions and $(\texttt{return}\ e_1)$ has result e_1 which is fed as an argument to e_2. This transformation is called (lunit) (the left identity monad law). It is only executed in *CHF**, if the expression is in a (monadic) reduction position in a thread. As a novel result we show that this transformation is an improvement also in all contexts. Finally, the transformation (drfork) that removes deterministic future calls and its inverse are analyzed.

An example for sequential and parallel improvements and illustrating the expressiveness of our approach is the parallelized fold of the leaf-elements in a tree, implemented in Concurrent Haskell (or *CHF**), see Fig. 1[1]. Depending on the functions f, g, this could be the sum of numbers in the leaves, or the search for a number in the tree. For simplicity, we assume that the tree `someTree` is given as a finite and fully evaluated tree, and that the functions f, g are strict in all of their arguments. The program `mainPure` is the pure version, `mainFut` is a very similar parallelized *CHF**-version using futures, and `mainMVar` is the Concurrent Haskell version with explicit synchronization using MVars.[2] Let us compare the interleaved and parallel reduction lengths of `mainPure`, `mainFut`, and `mainMVar`. An analysis and informal reasoning shows that `mainPure` is a (sequential) improvement of `mainFut` and `mainMVar`. For trees that are not too small, again informal reasoning shows that `mainFut` is a parallel improvement over `mainMVar`, which in turn is a parallel improvement over `mainPure`. For very

[1] We use Haskell's do-notation that is a shorthand for a sequence of `>>=`-expressions

[2] We remark that `calcMVar` evaluates the tree in a strict manner, while for `calcPure` and `calcFut` this depends on the function f.

small trees, `mainPure` is a parallel improvement of the two others, due to the overhead of bind and additional `seq`-evaluation steps. Our paper and results will provide methods and techniques to prove some of these improvements formally (see the Conclusion).

Results. The *results* of this paper are:

- We develop resource models for runtime in a concurrent call-by-need calculus and introduce the corresponding notions of improvements (Sect. 3)
- For all deterministic reduction rules of the operational semantics and for additional 15 program transformations and (drfork) (which are known as correct), we show that they are sequential improvements and for the same set of 15 transformations and the inverse of (drfork) we show that these are also parallel improvements (see Theorem 4.3). The transformations include functional transformations like partial evaluation, garbage collection, unique copying, and common subexpression elimination, but also transformations which pre-evaluate monadic computations or remove them (like deterministic thread elimination), which may change the runtime and sequence of actions.
- A corollary is that the translation of *CHF** to an abstract-machine-friendly representation is an improvement equivalence (Theorem 4.5).
- We develop proof techniques to show the improvement property including a diagram method and the notion of thread-normalized reductions (see Sect. 5, in particular Sect. 5.2).

Outline. In Sect. 2 we introduce the syntax of the calculus *CHF**, its operational semantics, and the contextual semantics. In Sect. 3 the resource models of sequential and parallel improvements are defined. In Sect. 4 we summarize our results for specific program transformations. In Sect. 5 we explain our proof technique of using reduction diagrams in conjunction with thread-normalized reductions (see Definitions 5.7 and 5.12 and Lemma 5.13). We conclude in Sect. 6 by first reconsidering the example programs of Fig. 1 and applying our obtained improvement results to them. We then discuss related and previous work on improvements, and finally we summarize our results and discuss potential further work. Missing proofs can be found in the appendix.

2 The Process Calculus *CHF**

We present the syntax, the type system, and the operational semantics of the program calculus *CHF** which models a core language of Concurrent Haskell extended by futures. We assume a partitioned set of *data constructors* c such that each family represents a type T. We assume that the data constructors of T are $c_{T,1}, \ldots, c_{T,|T|}$ and that 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 by [28], has processes on the top-layer which may have expressions (the second layer) as subterms. Processes

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

Fig. 2. Syntax of expressions, processes, and types

$$
\begin{array}{lll}
P_1 \mid P_2 \equiv P_2 \mid P_1 & (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3) & (\nu x.P_1) \mid P_2 \equiv \nu x.(P_1 \mid P_2) \\
\nu x_1.\nu x_2.P \equiv \nu x_2.\nu x_1.P & P_1 \equiv P_2 \text{ if } P_1 =_\alpha P_2 & \text{if } x \notin FV(P_2)
\end{array}
$$

Fig. 3. Structural congruence $\equiv$

$$
\begin{array}{l}
\mathbb{E} \in \mathit{ECtxt} ::= [\cdot] \mid (\mathbb{E}\,e) \mid \mathtt{case}\,\mathbb{E}\,\mathtt{of}\,\mathit{alts} \mid \mathtt{seq}\,\mathbb{E}\,e \quad \mathbb{M} \in \mathit{MCtxt} ::= [\cdot] \mid \mathbb{M} \gg\!= e \\
\mathbb{F} \in \mathit{FCtxt} ::= \mathbb{E} \mid \mathtt{takeMVar}\ \mathbb{E} \mid \mathtt{putMVar}\ \mathbb{E}\ e \quad \mathbb{D} \in \mathit{PCtxt} ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D} \\
\mathbb{L} \in \mathit{LCtxt} ::= x \Leftarrow \mathbb{M}[\mathbb{F}] \mid (x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[y] \mid y = \mathbb{E}_1), \\
\qquad\qquad \text{s.t. } \mathbb{E}_i \neq [\cdot] \text{ for } 2 \leq i \leq n \\
\widehat{\mathbb{L}} \in \widehat{\mathit{LCtxt}} ::= x \Leftarrow \mathbb{M}[\mathbb{F}] \mid (x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[y] \mid y = \mathbb{E}_1), \\
\qquad\qquad \text{s.t. } \mathbb{E}_i \neq [\cdot] \text{ for } 1 \leq i \leq n
\end{array}
$$

Fig. 4. *PCtxt*, *MCtxt*,*ECtxt*, *FCtxt*, *LCtxt*, and $\widehat{\mathit{LCtxt}}$-contexts.

Monadic Computations

(sr,lunit) $y \Leftarrow \mathbb{M}[\mathtt{return}\ e_1 \gg\!=\ e_2] \xrightarrow{sr} y \Leftarrow \mathbb{M}[e_2\ e_1]$

(sr,tmvar) $y \Leftarrow \mathbb{M}[\mathtt{takeMVar}\ x] \mid x\,\mathbf{m}\,e \xrightarrow{sr} y \Leftarrow \mathbb{M}[\mathtt{return}\ e] \mid x\,\mathbf{m}-$

(sr,pmvar) $y \Leftarrow \mathbb{M}[\mathtt{putMVar}\ x\ e] \mid x\,\mathbf{m}- \xrightarrow{sr} y \Leftarrow \mathbb{M}[\mathtt{return}\ ()] \mid x\,\mathbf{m}\,e$

(sr,nmvar) $y \Leftarrow \mathbb{M}[\mathtt{newMVar}\ e] \xrightarrow{sr} \nu x.(y \Leftarrow \mathbb{M}[\mathtt{return}\ x] \mid x\,\mathbf{m}\,e)$, where x is fresh

(sr,fork) $y \Leftarrow \mathbb{M}[\mathtt{future}\ e] \xrightarrow{sr} \nu z.(y \Leftarrow \mathbb{M}[\mathtt{return}\ z] \mid z \Leftarrow e)$, where z is fresh

(sr,unIO) $y \Leftarrow \mathtt{return}\ e \xrightarrow{sr} y = e$, if the thread is not the main-thread

Functional Evaluation

(sr,cp) $\widehat{\mathbb{L}}[x] \mid x = v \xrightarrow{sr} \widehat{\mathbb{L}}[v] \mid x = v$, if v is an abstraction or a variable

(sr,cpcxa) $\widehat{\mathbb{L}}[x] \mid x = c\ e_1 \ldots\ e_n$
$\xrightarrow{sr} \nu y_1, \ldots y_n.(\widehat{\mathbb{L}}[x] \mid x = c\ y_1\ \ldots\ y_n \mid y_1 = e_1 \mid \ldots \mid y_n = e_n)$
if c is a constructor, or `return`, $\gg\!=$, `takeMVar`, `putMVar`, `newMVar`, or `future`; and in addition some e_i is not a variable. Only the non-variables e_j are abstracted

(sr,cpcxb) $\widehat{\mathbb{L}}[x] \mid x = c\ y_1 \ldots\ y_n \xrightarrow{sr} (\widehat{\mathbb{L}}[c\ y_1\ \ldots\ y_n] \mid x = c\ y_1\ \ldots\ y_n)$
if c is a constructor, or `return`, $\gg\!=$, `takeMVar`, `putMVar`, `newMVar`, or `future`

(sr,mkbinds) $\mathbb{L}[\mathtt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \mathtt{in}\ e]$
$\xrightarrow{sr} \nu x_1 \ldots x_n.(\mathbb{L}[e] \mid x_1 = e_1 \mid \ldots \mid x_n = e_n)$

(sr,lbeta) $\mathbb{L}[((\lambda x.e_1)\ e_2)] \xrightarrow{sr} \nu x.(\mathbb{L}[e_1] \mid x = e_2)$

(sr,case) $\mathbb{L}[\mathtt{case}_T\ c\,e_1 \ldots e_n\ \mathtt{of} \ldots (c\,y_1 \ldots y_n \to e) \ldots]$
$\xrightarrow{sr} \nu y_1 \ldots y_n.(\mathbb{L}[e] \mid y_1 = e_1 \mid \ldots \mid y_n = e_n])$, if $n > 0$

(sr,case) $\mathbb{L}[\mathtt{case}_T\ c\ \mathtt{of}\ \ldots (c \to e) \ldots] \xrightarrow{sr} \mathbb{L}[e]$

(sr,seq) $\mathbb{L}[(\mathtt{seq}\ v\ e)] \xrightarrow{sr} \mathbb{L}[e]$, if v is a functional value

Closure: If $P_1 \equiv \mathbb{D}[P_1']$, $P_2 \equiv \mathbb{D}[P_2']$, and $P_1' \xrightarrow{sr} P_2'$ then $P_1 \xrightarrow{sr} P_2$

We assume capture avoiding reduction for all reduction rules.

Fig. 5. Standard reduction rules

and expressions are defined by the grammars in Fig. 2 where *Var* is a countably-infinite set of variables, denoted with u, w, x, y, z.

Parallel processes are formed by parallel composition "$\mid$", ν-binders restrict the scope of variables, 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 $x = e$ represent the global heap of shared expressions, where x is called a *binding variable*. For a process P, a variable x is an *introduced variable* if x is a future, a name of an MVar, or a binding variable. An introduced variable is visible to the whole process unless its scope is restricted by a ν-binder, i.e. in $Q \mid \nu x.P$ the scope of x is P. 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 $\texttt{case}_T\ e\ \texttt{of}\ \mathit{Alts}$ 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. We use $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ for the `case`-expression $\texttt{case}_{Bool}\ e\ \texttt{of}\ (\texttt{True} \to e_1)\ (\texttt{False} \to e_2)$. 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 abbreviate (parts of) `letrec`-environments as *Env*, and thus e.g. write $\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ e$. *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. 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.

Variable binders are introduced by abstractions, `letrec`-expressions, `case`-alternatives, and by $\nu x.P$. This induces a notion of free and bound variables, α-renaming, and α-equivalence (denoted by $=_\alpha$). Let $FV(P)$ ($FV(e)$, resp.) be the free variables of process P (expression e, resp.). For a set $x_1{=}e_1, \ldots, x_n{=}e_n$ of `letrec`-bindings or a sequence of bindings $x_1{=}e_1 \mid \ldots \mid x_n{=}e_n$, let $LV(x_1 = e_1, \ldots, x_n = e_n)$ and $LV(x_1 = e_1 \mid \ldots \mid x_n = e_n)$ the set of let-bound variables $\{x_1, \ldots, x_n\}$. We assume the *distinct variable convention* to hold: free variables are distinct from bound variables, and bound variables are pairwise distinct. We

assume that reductions implicitly perform α-renaming to obey this convention. In Fig. 3 structural congruence $\equiv$ of processes is defined.

The set of monomorphic types of constructor c is denoted as $\texttt{types}(c)$. The syntax of types *Typ* is given in Fig. 2 where $(\texttt{IO}\ \tau)$ stands for a monadic action with return type τ, $(\texttt{MVar}\ \tau)$ stands for an `MVar`-reference with content type τ, and $\tau_1 \to \tau_2$ is a function type. We assume that every variable is explicitly typed by a global typing function Γ, s.t. $\Gamma(x)$ is the type of variable x. The notation $\Gamma \vdash e :: \tau$ means that type τ can be derived for expression e using the global typing function Γ, and for processes, the notation $\Gamma \vdash P :: \texttt{wt}$ means that the process P can be well-typed using the global typing function Γ. We omit the (standard) monomorphic typing rules, but emphasize some special restrictions: $x \Leftarrow e$ is well-typed, if $\Gamma \vdash e :: \texttt{IO}\ \tau$, and $\Gamma \vdash x :: \tau$, the first argument of `seq` must not be an `IO`- or `MVar`-type. A process P is *well-typed* iff P is well-formed and $\Gamma \vdash P :: \texttt{wt}$ holds. An expression e is *well-typed* with type τ (written as $e :: \tau$) iff $\Gamma \vdash e :: \tau$ holds.

We recall the operational semantics of *CHF**, which is a small-step reduction relation called *standard reduction*. The presentation here is analogous to [28] with the difference that we use the two rules (sr,cpcxa) and (sr,cpcxb) instead of the rule (sr,cpcx). This modification does not change the semantics as we show in Theorem A.1. *Successful processes* are the successful outcomes of the standard reduction. They capture the behavior that termination of the main-thread implies termination of the whole program. A well-formed process P is *successful*, if $P \equiv \nu x_1 \ldots \nu x_n.(x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ e \mid P')$. We permit standard reductions only for well-formed processes which are not successful. A context is a process or an expression with a hole $[\cdot]$. We assume that the hole $[\cdot]$ is typed and carries a type label, which we sometimes write as $[\cdot^\tau]$. The typing rules are accordingly extended by the axiom for the hole: $\Gamma \vdash [\cdot^\tau] :: \tau$. Given a context $C[\cdot^\tau]$ and an expression $e :: \tau$, $C[e]$ denotes the result of replacing the hole in C with expression e. Since our syntax has different syntactic categories, we require different contexts (see Fig. 4): (i) process contexts that are processes with a hole at process position, (ii) expression contexts that are expressions with a hole at expression position, and (iii) process contexts with an expression hole. The standard reduction rules use process contexts (together with the structural congruence) to select some components for the reductions. In general, these components are a single thread, or a thread and a (filled or empty) MVar, or a thread and a set of bindings (which are referenced and used by the selected thread). Analogous to [28], we define *monadic contexts MCtxt*, *expression evaluation contexts ECtxt*, *forcing contexts FCtxt*, and the functional evaluation contexts *LCtxt*, $\widehat{LCtxt}$ (see Fig. 4) for modeling the call-by-need (concurrent) standard reduction. The *standard reduction rules* are given in Fig. 5 and with the closure w.r.t. *PCtxt*-contexts and $\equiv$, they define the standard reduction $\xrightarrow{sr}$. With $\xrightarrow{sr,+}$ we denote the transitive closure of $\xrightarrow{sr}$, and with $\xrightarrow{sr,*}$ we denote the reflexive-transitive closure of $\xrightarrow{sr}$. The small-step reduction rules consist of rules to perform monadic computations, and of rules to perform functional evaluation on expressions. The rule (sr,lunit) implements monadic sequencing for the operator $\gg\!=$. The rules (sr,tmvar) and

(sr,pmvar) perform a `takeMVar`- or a `putMVar`-operation on a filled (or empty, resp.) MVar. The rule (sr,nmvar) creates a new filled MVar. The rule (sr,fork) spawns a new future for a concurrent computation. The rule (sr,unIO) binds the result of a monadic computation to a functional binding, i.e. the value of a concurrent future becomes accessible.

The rule (sr,cpcxa) shares the (non-variable) arguments of constructor applications (and monadic expressions) which occur in a needed binding $x = e$. The rules (sr,cp), and (sr,cpcxb) inline a needed binding $x = e$ where e must be an abstraction, a variable, a flat constructor application or a flat monadic expression. The rule (sr,mkbinds) moves the bindings of a `letrec`-expression into the global heap bindings. The rule (sr,lbeta) is the sharing variant of β-reduction. The (sr,case)-reduction reduces a `case`-expression, where – if the scrutinee is not a constant – bindings are created to implement sharing. The (sr,seq)-rule evaluates a `seq`-expression and replaces it with the second argument provided the first argument is a functional value.

We define the *redex* of the reduction rules. For (sr,lunit), (sr,tmvar), (sr,pmvar), (sr,nmvar), (sr,fork), it is the monadic expression in the context $\mathbb{M}$. For rule (sr,unIO), the redex is $y \Leftarrow \texttt{return}\ e$, for (sr,mkbinds), (sr,lbeta), (sr,case), (sr,seq), the redex is the functional expression in the context $\mathbb{L}$, and for (sr,cp), (sr,cpcxa), (sr,cpcxb) the redex is the variable x in the context $\widehat{\mathbb{L}}$.

We briefly recall the notion of contextual equivalence with may- and should-convergence as observations (see [28]). 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. As observations we use may- and should-convergence:

Definition 2.1. *A process P* may-converges *(written as $P{\downarrow}$), iff it is well-formed and reduces to a successful process, i.e. $P{\downarrow}$ iff P is well-formed and $\exists P' : P \xrightarrow{sr,*} P' \wedge P'$ is successful. 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, i.e. $P{\Downarrow}$ iff P is well-formed and $\forall P' : P \xrightarrow{sr,*} P' \implies P'{\downarrow}$. If P is not should-convergent then we say P* may-diverges *written as $P{\uparrow}$.*

We write $P{\downarrow}P'$ (or $P{\uparrow}P'$, resp.) if $P \xrightarrow{sr,*} P'$ and P' is successful (or must-divergent, resp.).

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 $\gamma \subseteq \sim_c$.

We sometimes attach further information to reduction or transformation arrows, e.g. $\xrightarrow{sr,a,k}$ means k sr-reductions of kind a; we use $*$ and $+$ to denote the reflexive-transitive and the transitive closure, respectively. The notation $\xrightarrow{a \vee b}$ means a reduction of kind a or of kind b.

3 Sequential and Parallel Improvements in *CHF**

In deterministic programming languages, a program transformation is an improvement iff it is correct and it does not increase the length of reduction sequences in any context (but usually decreases the reduction length in many cases). Here the "length" of reduction sequences may also cover only essential reductions steps instead of all steps. Investigations of improvements and techniques to show that transformations are improvements for deterministic functional program calculi with call-by-need evaluation can be found in [20, 11, 33, 35]. In this paper we are concerned with a concurrent functional language and thus we require an adapted definition of improvement for this scenario. Concurrency as given by the evaluation of *CHF**-programs has two differences compared to the programs in purely deterministic functional languages: i) evaluation is non-deterministic and thus may lead to different results; ii) evaluation of concurrent threads gives rise to parallel execution of threads (on several processors) and thus speeds up the execution of programs.

We consider two improvement relations covering both aspects of concurrent evaluation. The first one can be seen as a single-processor model while the other one is adapted for a multi-processor scenario. In order to count the time required for evaluations in *CHF**, we will count reduction steps where we consider two forms of evaluations: sequences of interleaved reductions from the concurrent threads (called *sequential reductions*), and sequences of *parallel reductions*, where threads run in parallel. To restrict the length measure to essential reduction steps, we use sets A of *reduction kinds* from Fig. 5, where only the name is of interest and where we abstract from the exact expressions and application positions. Let A_{all} be the set of all reduction kinds, and let $A_{cp} := \{(sr, cp), (sr, cpcxa), (sr, cpcxb), (sr, mkbinds)\}$. The main set of reductions that we use is the set $A_{noncp} := A_{all} \setminus A_{cp}$, however, for some of our results, we also use other subsets of A_{all}.

We argue why considering the number of reductions in A_{noncp} and thus omitting A_{cp}-reductions is sufficient. Abstract machines like variants of the Sestoft machines [38] usually do not need (cp) for variables, nor (cpcxa), i.e. copying variables and abstracting functional or monadic values, since this is built-in due to the restricted structure of machine expressions. The other reduction kinds (cp), (cpcxb), and (mkbinds) only occur as follows (if reduction steps are viewed per thread): Several (mkbinds) are always followed by a A_{noncp}-reduction or by a (cp) which copies an abstraction and then a A_{noncp}-reduction. The same holds for (cpcxb). The extra effort is at most the size of the initial process. Summarizing, the number of A_{noncp}-steps is a characteristic factor indicating the time required for evaluation. We omit an explicit detailed analysis in this paper, which could be done in a similar way to the analysis of [33] which was performed for a deterministic setting.

3.1 Sequential Improvements

A sequential A-improvement improves the length of minimal and successful reduction sequences w.r.t. the reduction kinds in A:

Definition 3.1. *Let P be a well-formed process with $P{\downarrow}$, $A \subseteq A_{all}$, and Red be a successful reduction sequence of P. Let $srnr_A(Red)$ be the number of A-reductions occurring in Red. We define $srnr_A(P) := \min\{srnr_A(Red) \mid Red$ is a successful standard reduction of $P\}$.*

Let P_1 and P_2 be two well-formed processes with $P_1{\downarrow}, P_2{\downarrow}$ and $P_1 \sim_c P_2$. If $\forall \mathbb{D} \in PCtxt : srnr_A(\mathbb{D}[P_1]) \leq srnr_A(\mathbb{D}[P_2])$, then P_1 sequentially A-improves *P_2, written $P_1 \preceq_A P_2$. If $P_1 \preceq_A P_2$ and $P_2 \preceq_A P_1$, then we say P_1, P_2 are* improvement-equivalent w.r.t. *A (and interleaved reduction). A program transformation $\xrightarrow{PT}$* is a sequential A-improvement *if $P_1 \xrightarrow{PT} P_2$ implies that P_2 sequentially A-improves P_1 for all processes P_1, P_2. We say that $\xrightarrow{PT}$ is a* sequential A-improvement equivalence *iff $\xrightarrow{PT}$ and $\xrightarrow{PT^-}$ (the inverse of $\xrightarrow{PT}$) are both sequential A-improvements.*

Sequential A-improvements are related to counting the length of reductions in the deterministic calculus LR by [36], where the main measure only counts (lbeta), (case), and (seq)-reductions. It is an adaptation of the improvement notions in [33, 35] to our concurrent, non-deterministic standard reduction.

For motivating our notion of improvement and to detail the abstract model in the introduction, we first consider processes P, P' s.t. P' is some (conservatively) parallelized version of a pure program P. Then in general there is only one reduction length for all possible reductions. Hence in this case taking the minimum has no effect and improvements are the same as in the deterministic case. In the more general case, we motivate that comparing the minimal reduction lengths covers the intuitive notion also in the case of a non-deterministic program, and if there are different evaluations leading to incomparable results. To be more concrete, let P_2 be:

$$P_0 \mid w_1 \Leftarrow \texttt{seq}\ x_1\ (\texttt{putMVar}\ z\ x_1) \\ \mid w_2 \Leftarrow \texttt{seq}\ x_2\ (\texttt{putMVar}\ z\ x_2) \mid z\,\mathbf{m} - \mid x_1 = e_1 \mid x_2 = e_2$$

where e_1 evaluates to 1 and e_2 evaluates to 2 (perhaps also generating some global bindings). Suppose the evaluation of e_1 is shorter than that of e_2, and $e_2' \sim_c e_2$ is an expression that requires strictly more reduction steps than e_2 to evaluate to 2. Let P_2' be P_2, where e_2 is replaced by e_2'.

Using the results from [28, 29], we see that $P_2' \sim_c P_2$. The first impression is that P_2' and P_2 are equivalent w.r.t. improvement, since the standard reduction that prefers to put e_1 into the MVar z may in both cases be chosen for comparing the number of reductions, since it has less reductions. So let us conjecture that $P_2' \preceq P_2$. However, since the property of having shorter reductions must hold in any surrounding context $\mathbb{D}$, we can also choose a process context including an expression e_3 with a very long computation as follows: $\mathbb{D} = y \Leftarrow \texttt{takeMVar}\ z \mid [\cdot]$ $\mid u \overset{\mathsf{main}}{\Longleftarrow} \texttt{if}\ y == 1\ \texttt{then seq}\ x\ (\texttt{return}\ x)\ \texttt{else return}\ 0 \mid x = e_3$

Comparing $\mathbb{D}[P_2']$ and $\mathbb{D}[P_2]$ shows that the reduction sequences that evaluate e_1 are now the longer ones and the evaluation of e_2 determines the minimum, hence $srnr(\mathbb{D}[P_2']) > srnr(\mathbb{D}[P_2])$, and our conjecture is false. This shows that our definition using outer contexts and the minimal number of reductions is sensible for the different non-deterministic possibilities of reductions.

3.2 Parallel Improvements

For measuring the duration of parallel evaluations of processes by their lengths, we first have to precisely define the notion of parallel evaluation (or parallel reduction sequences):

Definition 3.2 (Parallel evaluation). *Let P be a well-formed process and let us assume w.l.o.g. that it is in ν-prenex form $\nu x_1 \dots x_n.P_0$. Then a* parallel reduction*, written as $P \xrightarrow{srp} P'$, is the result of several (at least one) sr-reduction steps at once, provided there is no interference between the (syntactic) effects. This can also be defined as a sequence of n sr-reductions, where every permutation of the reduction sequence is executable and leads to the same resulting expression (up to α-renaming and structural congruence). The exact details of "no interference" are as follows:*

1. *For the monadic computations and for (cpcxa), (mkbinds), (lbeta), (case), and (seq) in Fig. 5, the parallel reduction is $P_0 = R \mid P_{0,R} \to R' \mid P'_{0,R}$ where R is the redex of the sr-reductions, and $P_{0,R} \to P'_{0,R}$ is a parallel reduction of the rest.*
2. *For (cp), the reduction is $R \mid (x = v) \mid P_{0,R} \to R' \mid (x = v) \mid P'_{0,R}$ where R is the (cp)-redex, and $(x = v) \mid P_{0,R} \to (x = v) \mid P'_{0,R}$ is a parallel reduction.*
3. *For (cpcxb), $R \mid x = c\ y_1 \dots y_n \mid P_{0,R} \to R' \mid x = c\ y_1 \dots y_n \mid P'_{0,R}$ is the reduction where R is the (cpcxb)-redex and $x = c\ y_1 \dots y_n \mid P_{0,R} \to x = c\ y_1 \dots y_n \mid P'_{0,R}$ is a parallel reduction.*

A parallel reduction sequence is successful *if the last process is successful. The number of parallel single reductions in a parallel reduction step is limited by the number of available processors, sometimes denoted by the number N.*

Note that in a parallel reduction step, the following observations are valid: i) There is at most one sr-reduction per thread. ii) Single functional reduction steps may be triggered by several threads. iii) If several threads try to access the same MVar, then conflicts occur.

There is no standard form of a parallel reduction sequence. Only for a successful reduction sequence without MVar-accesses, i.e. a deterministic one, and if an unbounded number of processors is available, an eager scheduling leads to the shortest parallel reduction. But even this parallel reduction sequence is not unique and in general not optimal w.r.t. the work.

Definition 3.3. *Let P be a well-formed process with $P{\downarrow}$, let A be a set of reduction kinds, and let $N \in \{1, 2, \dots\} \cup \{\infty\}$ be the number of available processors.*

For a parallel reduction sequence Red, let $srnrp_A^N(Red)$ *be the number of parallel reduction steps for at most N processors that contain an A-reduction. If* $N = \infty$*, then we may omit the superscript N. Let* $srnrp_A^N(P)$*, the* parallel number of A-steps*, be the minimum of* $\{srnrp_A^N(Red) \mid Red$ *is a successful parallel reduction with at most N processors of* $P\}$*.*

For well-formed P_1, P_2 *with* $P_1{\downarrow}, P_2{\downarrow}$*,* $P_1 \sim_c P_2$*, we say* P_1 parallel improves P_2 *w.r.t. A and N processors; notation* $P_1 \preceq_{p,N,A} P_2$*, iff* $\forall \mathbb{D} \in PCtxt :$ $srnrp_A^N(\mathbb{D}[P_1]) \leq srnrp_A^N(\mathbb{D}[P_2])$*. (We mainly use* A_{noncp}*.) If* $P_1 \preceq_{p,N,A} P_2$ *and* $P_2 \preceq_{p,N,A} P_1$*, then we say* P_1, P_2 *are* improvement-equivalent w.r.t. A, N *and parallel reduction. A program transformation* $\xrightarrow{PT}$ *is a* parallel improvement w.r.t. A and N processors*, iff* $P_2 \xrightarrow{PT} P_1$ *implies* $P_1 \preceq_{p,N,A} P_2$ *for all processes* P_1, P_2*, and it is a* parallel improvement equivalence w.r.t. A, N *iff* $P_2 \xrightarrow{PT} P_1$ *implies that* P_1 *and* P_2 *are improvement-equivalent w.r.t* A *and* N*.* □

Remark 3.4. In *CHF** there is an exponential upper bound for the acceleration by parallelizing: If P is a process that is started from a single thread. Then $srnr(P) < 2^{srnrp(P)+1}$. The reason is that every parallel reduction step can at most double the number of threads and this doubling is done by (fork)-reductions. Hence the overall number of reduction steps is at most $1 + 2 + \ldots + 2^{srnrp(P)} < 2^{srnrp(P)+1}$.

4 Proven Improvements

In this section we summarize our concretely obtained results on sequential and parallel improvements in the process calculus *CHF**. For readability, the proofs and the used proof techniques are deferred to later sections.

We define several program transformations for which we have checked whether they are sequential and/or parallel improvements. In Fig. 6 we define general, surface, and top contexts. In Fig. 7 the program transformations are defined where the first part are generalizations of some standard reductions. The rules (dtmvar) and (dpmvar) are variants of (sr,tmvar) and (sr,pmvar) where the side conditions ensure that the MVar-access is deterministic. The rule (drfork) removes a future-operation and thus performs thread elimination provided that the corresponding computation does not access the storage (i.e. any MVar). The three rules named (gc) represent a form of garbage collection, where the first rule operates on the process level and allows the removal of global bindings and MVars, while the other rules operate on the expression level and allow to remove (parts of) letrec-environments. The rule (ucp) means "unique copying" and allows inlining of an expression which is referenced only once. The representation of the rule is split into two parts (two rules (ucpt) and one rule (ucpd)) where (ucpt) is not applied below an abstraction and (ucpd) is always applied inside an abstraction. The two variants of rule (ucpt) are: the first one operates on the process level and inlines a global binding, while the second one operates on the expression level and inlines a (local) letrec-binding. Finally, the transformation (cse) performs common subexpression elimination where the first rule replaces

$\mathbb{C} \in \mathit{CCtxt}$:= *general contexts: process contexts with the hole at expression position.*
$\mathbb{S} \in \mathit{SCtxt}$:= *surface contexts: CCtxt with the hole not inside an abstraction.*
$\mathbb{T} \in \mathit{TCtxt}$:= *top contexts: SCtxt with the hole not inside a* `case`*-alternative.*

Fig. 6. Context classes for transformations

a duplicated expression by a reference to the copy, and the other rules remove a duplicated environment (in a local letrec or as part of the global bindings).

Definition 4.1. *In Fig. 7 several program transformations are defined using general, surface, and top contexts defined in Fig. 6. The transformations are assumed to be closed w.r.t. $\equiv$ and w.r.t. PCtxt-contexts and in all rules only instances that do not violate the scoping are permitted.*

Remark 4.2. There are sufficient criteria for the applicability of (dtmvar) and (dpmvar), for example, if $\mathbb{D} = [\cdot]$, or if neither $\mathbb{M}$, e nor $\mathbb{D}$ contain occurrences of x, or if $\nu x.\mathbb{D}[\mathbb{M}[\cdot]]$ is closed and $\mathbb{D}$ does not contain any `takeMVar` nor `putMVar`.

Theorem 4.3. *In Table 1 we summarize the established results for the considered concrete program transformations concerning the property of being sequential improvements and being parallel improvements. The results also imply:*

- *(ucp) is a sequential A-improvement equivalence for all A with $A \subseteq A_{noncp}$,*
- *(gc) is a sequential A-improvement equivalence for all A with $A \subseteq A_{all} \setminus \{mkbinds\}$,*
- *(ucp) is a parallel A-improvement equivalence for all A with $A \subseteq A_{noncp}$,*
- *(gc) is a parallel A-improvement equivalence for all A with $A \subseteq A_{noncp}$.*

Proof. We defer the proofs to later sections or to the appendix. However, at this point we provide pointers to the proofs: The results on sequential A-improvements are proved in Theorem 5.4 (for (sr,a)-transformations), in Proposition B.11 for (lbeta), (case), and (seq), in Proposition B.10 for (mkbinds), in Proposition B.6 for (cp), in Propositions B.1 for (gc) and in Proposition B.2 for $(\mathrm{gc})^-$, in Proposition B.3 for (ucp) and in Propositions B.5, B.4 for $(\mathrm{ucp})^-$, in Proposition B.7 for (cpcxa), in Proposition B.9 for (cpcxb), in Proposition B.13 for (cse), in Proposition B.14 for (lunit), in Proposition B.15 for (nmvar), in Proposition B.16 for (dtmvar) and (dpmvar), in Proposition B.18 for (drfork). The results on parallel improvements are proved in Theorem 5.19 and Proposition B.18.

As a further topic which also motivates the analysis of transformation (ucp), let us consider a translation into code for an abstract machine. [27] provides a concurrent abstract machine to execute CHF-programs which extends Sestoft's machine [38], which was designed for call-by-need evaluation of purely functional expressions. The abstract machine is restricted to *simplified* expressions of *CHF**: In a simplified expression several argument positions are restricted to be variables only. We indicate the restrictions: $(e\ x)$, $(\texttt{seq}\ e\ x)$, $(c\ x_1 \ldots x_n)$, $x\ \mathbf{m}\ y$, $\texttt{return}\ x$, $x_1 \gg\!= x_2, \texttt{future}\ x$, $\texttt{takeMVar}\ x, \texttt{putMVar}\ x_1\ x_2, \texttt{newMVar}\ x$. The following definition shows how the restrictions can be achieved.

(lunit) $\mathbb{C}[\texttt{return}\ e_1 \gg\texttt{=}\ e_2] \xrightarrow{sr} \mathbb{C}[e_2\ e_1]$

(cp) $\mathbb{C}[x] \mid x = v \xrightarrow{sr} \mathbb{C}[v] \mid x = v$, if v is an abstraction or a variable

(cpcxa) $\mathbb{C}[x] \mid x = c\ e_1 \ldots e_n \xrightarrow{sr} \nu y_1, \ldots y_n.(\mathbb{C}[x] \mid x = c\ y_1 \ldots y_n \mid y_1{=}e_1 \mid \ldots \mid y_n{=}e_n)$, if c is a monadic operator or a constructor and some e_i is not a variable, and where only the non-variables e_j are abstracted

(cpcxb) $\mathbb{C}[x] \mid x = c\ y_1 \ldots y_n \xrightarrow{sr} (\mathbb{C}[c\ y_1 \ldots y_n] \mid x = c\ y_1 \ldots y_n)$, if c is a monadic operator or a constructor

(mkbinds) $\mathbb{C}[\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e] \xrightarrow{sr} \nu x_1 \ldots x_n.(\mathbb{C}[e] \mid x_1 = e_1 \mid \ldots \mid x_n = e_n)$

(lbeta) $\mathbb{C}[((\lambda x.e_1)\ e_2)] \xrightarrow{sr} \nu x.(\mathbb{C}[e_1] \mid x = e_2)$

(case) $\mathbb{C}[\texttt{case}_T\ c\, e_1 \ldots e_n\ \texttt{of} \ldots (c\, y_1 \ldots y_n \to e) \ldots] \xrightarrow{sr} \nu y_1 \ldots y_n.(\mathbb{C}[e] \mid y_1{=}e_1 \mid \ldots \mid y_n{=}e_n])$, if $n > 0$

(case) $\mathbb{C}[\texttt{case}_T\ c\ \texttt{of}\ \ldots (c \to e) \ldots] \xrightarrow{sr} \mathbb{C}[e]$

(seq) $\mathbb{C}[(\texttt{seq}\ v\ e)] \xrightarrow{sr} \mathbb{C}[e]$, if v is a functional value

(dtmvar) $\nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{takeMVar}\ x] \mid x\, \mathbf{m}\, e] \to \nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{return}\ e] \mid x\, \mathbf{m}\, -]$ if for all $\mathbb{D}' \in PCtxt$ and $\xrightarrow{sr,*}$-sequences starting with $\mathbb{D}'[\nu x.(\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{takeMVar}\ x] \mid x\, \mathbf{m}\, e])]$ the first execution of any ($\texttt{takeMVar}\ x$)-operation takes place in the y-thread.

(dpmvar) $\nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{putMVar}\ x\ e] \mid x\, \mathbf{m}\, -] \to \nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{return}\ ()] \mid x\, \mathbf{m}\, e]$ if for all $\mathbb{D}' \in PCtxt$ and $\xrightarrow{sr,*}$-sequences starting with $\mathbb{D}'[\nu x.(\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{putMVar}\ x\ e] \mid x\, \mathbf{m}\, -])]$ the first execution of any ($\texttt{putMVar}\ x\ e'$)-operation takes place in the y-thread.

(drfork) $\mathbb{C}[\texttt{future}\ e] \to \mathbb{C}[e]$ if for all $\mathbb{D} \in PCtxt$ and $\xrightarrow{sr,*}$-sequences starting with $\mathbb{D}[\mathbb{C}[\texttt{future}\ e]]$ the threads, started with $\texttt{future}\ e$, never will execute an action on an MVar.

(gc) $\nu x_1, \ldots, x_n.(P \mid \texttt{Comp}(x_1) \mid \ldots \mid \texttt{Comp}(x_n)) \to P$ if for all $i \in \{1, \ldots, n\}$: $\texttt{Comp}(x_i)$ is a binding $x_i = e_i$, an MVar $x_i\, \mathbf{m}\, e_i$, or an empty MVar $x_i\, \mathbf{m}\, -$, and $x_i \notin FV(P)$.

(gc) $\mathbb{C}[\texttt{letrec}\ Env\ \texttt{in}\ e] \to \mathbb{C}[e]$, if $FV(e) \cap LV(Env) = \emptyset$

(gc) $\mathbb{C}[\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ e] \to \mathbb{C}[\texttt{letrec}\ Env_2\ \texttt{in}\ e]$ if $(FV(Env_1, e)) \cap LV(Env_1) = \emptyset$

(ucpt) $\nu x.(\mathbb{S}[x] \mid x = e) \to (\mathbb{S}[e])$, if x does not occur in $\mathbb{S}, e$ and $\mathbb{S}$ does not bind x

(ucpt) $\mathbb{S}_1[\texttt{letrec}\ x = e, Env\ \texttt{in}\ \mathbb{S}_2[x]] \to \mathbb{S}_1[\texttt{letrec}\ Env\ \texttt{in}\ \mathbb{S}_2[e]]$ if x does not occur elsewhere and $\mathbb{S}_1$ and $\mathbb{S}_2$ do not bind x

(ucpd) $\mathbb{C}_1[\lambda y.\mathbb{C}_2[\texttt{letrec}\ x = e, Env\ \texttt{in}\ \mathbb{S}[x]]] \to \mathbb{C}_1[\lambda y.\mathbb{C}_2[\texttt{letrec}\ Env\ \texttt{in}\ \mathbb{S}[e]]]$ if x does not occur elsewhere and $\mathbb{C}_1$, $\mathbb{C}_2$, and $\mathbb{S}$ do not bind x

(ucp) $= (ucpd) \vee (ucpt)$

(cse) $\mathbb{C}[e] \mid x = e \to \mathbb{C}[x] \mid x = e$

(cse) $\mathbb{C}[\texttt{letrec}\ Env\ e] \mid Env' \to \mathbb{C}[e] \mid Env'$, if $\pi{\cdot}Env \sim_\alpha Env'$ for some permutation π that maps $LV(Env) \to LV(Env')$ and $LV(Env)$ is fresh for Env'

(cse) $x_1 = e_1 \mid \ldots \mid x_n = e_n \mid y_1 = e'_1 \mid \ldots \mid y_n = e_n \to x_1 = e_1 \mid \ldots \mid x_n = e_n$, if $\pi{\cdot}e_i \sim_\alpha \pi{\cdot}e'_i$ for the permutation π with $\forall i : \pi(x_i) = y_i$, $\pi(y_i) = x_i$ the variables x_i are not free in e_j for all j.

Closure: If $P_1 \equiv \mathbb{D}[P'_1]$, $P_2 \equiv \mathbb{D}[P'_2]$, and $P'_1 \xrightarrow{PT} P'_2$ then $P_1 \xrightarrow{PT} P_2$

Fig. 7. Program transformations. The permutation π in (cse) is a variable-to-variable (bijective) function on the expressions.

Table 1. Summary of improvement results

Transformation	is a sequential A-improvement for all A with ...	is a parallel A–improvement w.r.t. A and N for ...
(sr,a) for $a \notin \{\text{tmvar}, \text{pmvar}\}$	$A \subseteq A_{all}$	$A \subseteq A_{all}$ and every N
(lbeta), (case), and (seq)	$A \subseteq A_{all}$	$A \subseteq A_{noncp}$ and every N
(mkbinds)	$A \subseteq A_{all}$	$A \subseteq A_{noncp}$ and every N
(cp)	$A \subseteq A_{all}$	$A \subseteq A_{noncp}$ and every N
(gc)	$A \subseteq A_{all}$	$A \subseteq A_{noncp}$ and every N
$(\text{gc})^-$	$A \subseteq A_{all} \setminus \{\text{mkbinds}\}$	$A \subseteq A_{noncp}$ and every N
(ucp)	$A \subseteq A_{all} \setminus \{\text{cpcxa}\}$	$A \subseteq A_{noncp}$ and every N
$(\text{ucp})^-$	$A \subseteq A_{noncp}$	$A \subseteq A_{noncp}$ and every N
(cpcxa)	$A \subseteq A_{all} \setminus \{\text{mkbinds}\}$	$A \subseteq A_{noncp}$ and every N
(cpcxb)	$A \subseteq A_{all} \setminus \{\text{mkbinds}, \text{cpcxa}\}$	$A \subseteq A_{noncp}$ and every N
(cse)	$A \subseteq A_{noncp}$	$A \subseteq A_{noncp}$ and every N
(lunit)	$A \subseteq A_{noncp}$	$A \subseteq A_{noncp}$ and every N
(dtmvar), (dpmvar)	$A \subseteq A_{all}$	$A \subseteq A_{noncp}$ and every N
(drfork)	$A \subseteq A_{noncp}$	-
$(\text{drfork})^-$	-	$A \subseteq (A_{noncp} \setminus \{\text{fork}, \text{unIO}\})$

Definition 4.4. *The function σ translates processes into simplified processes. It is defined to be homomorphic over the term structure (e.g. $\sigma(P_1 \mid P_2) := \sigma(P_1) \mid \sigma(P_2)$, etc.) except for the cases:*

$$
\begin{array}{lll}
\sigma(e_1\ e_2) & := & \texttt{letrec}\ x = \sigma(e_2)\ \texttt{in}\ (\sigma(e_1)\ x) \\
\sigma(c\ e_1\ \ldots\ e_n) & := & \texttt{letrec}\ x_1 = \sigma(e_1), \ldots, x_n = \sigma(e_n)\ \texttt{in}\ c\ x_1\ \ldots\ x_n \\
 & & \textit{if } c \textit{ is a constructor, or a monadic operator} \\
\sigma(\texttt{seq}\ e_1\ e_2) & := & \texttt{letrec}\ x = \sigma(e_2)\ \texttt{in}\ \texttt{seq}\ \sigma(e_1)\ x \\
\sigma(x\, \mathbf{m}\, e) & := & x\, \mathbf{m}\, y \mid y = \sigma(e)
\end{array}
$$

The translation σ is equivalent to an iterated use of the inverse of (ucp).

Theorem 4.5. *The translation σ from full CHF* into the set of simplified CHF*-programs (the machine expressions) is an improvement equivalence w.r.t. every set $A \subseteq A_{noncp}$ of reduction kinds.*

Note that the restriction to $A \subseteq A_{noncp}$ is not really essential, since reductions like copying variables, (cpcxa), (cpcxb), (mkbinds) are not performed by the abstract machine due to the used data structures. There is also no essential difference in copying abstractions, since every copy of an abstraction is (in the same thread) immediately followed by a reduction (lbeta) or (seq), hence our measure is realistic for measuring the runtime of the machine.

5 Proofs and Proof Techniques

We now mainly explain our proof techniques which we have applied to derive the results on sequential and parallel improvements (as summarized in Theorem 4.3). For space reasons, most of the proofs are given in the appendix only.

5.1 Improvement Property of Standard Reductions

For proving any improvement property of a program transformation, we require that the transformation is correct. Correctness results for transformations under consideration can in many cases be imported from [28, 29] for the calculus *CHF* and by using the following equivalence between *CHF* and *CHF**:

Remark 5.1. The calculus *CHF* used by [28] and the calculus *CHF** introduced in this paper are equivalent w.r.t. may- and should-convergence, and also w.r.t. correctness of transformations, as proved in Appendix A. The reason to replace the CHF-rule (cpcx) by two rules (cpcxa), (cpcxb) is that these reductions lead to less conflicts in reasoning about transformations.

This allows the import of correctness results from previous work:

Theorem 5.2. *The standard reductions* (sr, fork), (sr, unIO), (sr, lunit), *and* (sr, nmvar), *and the transformations* (cp), (cpcxa), (cpcxb), (mkbinds), (lbeta), (case), (seq), (gc), (cse), (dtmvar), (dpmvar), (ucp), (cse), *and* (lunit) *are correct program transformations.*

Proof. This follows from the results in [28, Propositions 5.2, 5.6, 7.5 and Theorem 6.7] and from the equivalence of *CHF* and *CHF**. □

We show that the correct standard reductions are also sequential A-improvements for any set $A \subseteq A_{all}$ (see Definition 3.1).

Proposition 5.3. *Let* $A \subseteq A_{all}$, P *be a well-formed process with* $P{\downarrow}$ *and let* $P \xrightarrow{sr,a} P'$ *with* $a \notin \{pmvar, tmvar\}$. *Then* $srnr_A(P') \leq srnr_A(P)$.

Proof. If P' is successful, then the claim is trivial. By assumption, P is not (yet) successful, hence let *Red* be a reduction of P to a successful process, and let $P \xrightarrow{sr,b} P_1$ be the first step of *Red*. Note that $b \in \{pmvar, tmvar\}$ is possible. There are only three possibilities: reductions are equal, or they commute, which means they are in different threads, or P_1 is successful:

$$\begin{array}{ccc} P & \xrightarrow{sr,a} & P' \\ {\scriptstyle sr,b}\downarrow & & {\scriptstyle sr,b}\downarrow \\ P_1 & \xrightarrow{sr,a} & P_1' \end{array} \qquad \begin{array}{ccc} P & \xrightarrow{sr,a} & P' \\ {\scriptstyle sr,b}\downarrow & & {\scriptstyle sr,b}\downarrow \\ P_{1(\text{succ.})} & \xrightarrow{-b} & P'_{1(\text{succ.})} \end{array} \qquad \begin{array}{c} P \\ {\scriptstyle sr,b}\downarrow \quad \searrow {\scriptstyle sr,a} \\ P_1 \end{array}$$

Induction on the length of the reduction yields a reduction Red' of P' that is not greater for any A. Thus the minimum of reductions of P' w.r.t. A is smaller than the minimum for P. □

Note that there is no bound k s.t. $srnr_A(P) - srnr_A(P') \leq k$ in all situations of Proposition 5.3.

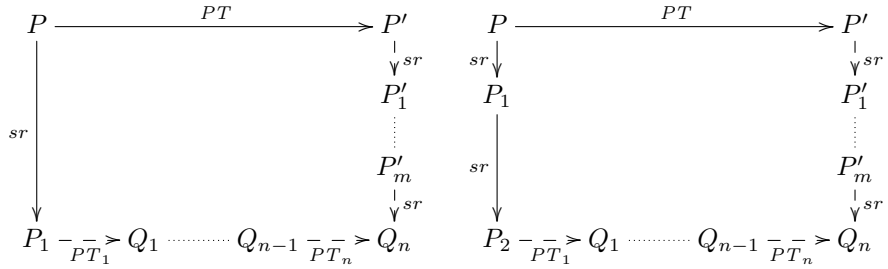
Theorem 5.4. *The standard reductions except for (tmvar), (pmvar) are improvements for any A.*

Proof. Let P be a process s.t. $P \xrightarrow{sr} P'$ and $\mathbb{D} \in PCtxt$ s.t. $\mathbb{D}[P]$ is well-formed. Then Proposition 5.3 can be applied to $\mathbb{D}[P]$ and $\mathbb{D}[P']$, since $\mathbb{D}[P] \xrightarrow{sr} \mathbb{D}[P']$, and thus the claim holds. □

5.2 Proving Sequential Improvements

We explain our proof technique for the case of sequential improvements. For a particular improvement $P' \preceq P$, we have a proof task for every $\mathbb{D}[P]$ and $\mathbb{D}[P']$. Since it is too hard to compute the explicit minimal lengths, and then to compare them, we show that for every reduction sequence of $\mathbb{D}[P]$ to a successful process, there is a shorter reduction sequence of $\mathbb{D}[P']$ to a successful process. This enables the use of constructive methods and the operational semantics that tell us how to modify a reduction sequence of $\mathbb{D}[P]$ to obtain a reduction sequence of $\mathbb{D}[P']$. Clearly, for this to work, the modifications from P to P' must be small and easily controllable. This is often the case for simple program transformations.

In order to increase the coverage of the method, we also require a standardization and a rearrangement of reductions sequences of $\mathbb{D}[P]$. The idea is to cut redundant parts of reduction sequences which do not contribute to the computation of the success. Since cutting makes the reduction sequences shorter, we see that it is sufficient for comparing the minimal reduction sequences to only consider the standardized reduction sequences, which we will call *thread-normalized reductions*. The construction of a (hopefully) shorter reduction sequence of $\mathbb{D}[P']$ from a (standardized) reduction sequence of $\mathbb{D}[P]$ will be done by the method of so-called forking diagrams. A forking diagram for transformation $\xrightarrow{PT}$ consists of a fork and join. A fork for transformation $\xrightarrow{PT}$ is of the form $P_1 \xleftarrow{sr} P \xrightarrow{PT} P'$ (or as an extended form $P_2 \xleftarrow{sr} P_1 \xleftarrow{sr} P \xrightarrow{PT} P'$). It describes an overlap between a standard reduction and a transformation step. A join is of the form $P_1 \xrightarrow{PT_1} Q_1 \dots Q_{n-1} \xrightarrow{PT_n} Q_n \xleftarrow{sr} P'_m \dots P'_1 \xleftarrow{sr} P'$ and it describes the reduction and transformation steps which can be used to close the overlap. Here the processes are in meta-notation, thus they may represent sets of processes. The graphical representation of the usual and extended form of forking diagram is as follows, where solid lines are used for the reductions and transformations of the fork, and dashed lines are used for the reductions and transformations of the join.



For $P_1 \xleftarrow{sr} P \xrightarrow{PT} P'$, a forking diagram is applicable iff there exist processes P'_i, Q_j with $P_1 \xrightarrow{PT_1} Q_1 \dots Q_{n-1} \xrightarrow{PT_n} Q_n \xleftarrow{sr} P'_m \dots P'_1 \xleftarrow{sr} P'$.

The forking diagrams are obtained by checking all cases for an overlap between standard reduction and transformations steps and then computing closing

reduction sequences. The diagrams are then used to construct a reduction sequence of $\mathbb{D}[P']$ from the given one for $\mathbb{D}[P]$. Of course, these diagrams must have a completeness property: every concrete overlap (within a thread-normalized and rearranged reduction sequence) has to be covered by at least one applicable diagram. We also allow forks where more than one $\xrightarrow{sr}$-reduction is given for P_1 (see the extended form above). However, since our standard reduction is nondeterministic, the standardization and rearrangement of reduction sequences is necessary, where the left-down reduction $P \xrightarrow{sr} P_1 \xrightarrow{sr} P_2$ are reductions triggered by the same thread. An example of such a diagram is the third (ucp)-diagram in Proposition B.3.

We define thread-normalized reduction sequences, which are, roughly speaking, those reduction sequences not containing unneeded functional evaluations. For reasoning on reduction sequences of minimal length, it is sufficient to consider thread-normalized reduction sequences.

Definition 5.5. *Let P be a process and $P \xrightarrow{sr} P'$ be a reduction step. Then the reduction step is* triggered *by thread y, if the reduction is within the context $\mathbb{D}[y \Leftarrow \mathbb{M}[\cdot]]$, in the context $\mathbb{D}[\widehat{\mathbb{L}}[\cdot]]$ or in the context $\mathbb{D}[\mathbb{L}[\cdot]]$, where $\widehat{\mathbb{L}}$, $\mathbb{L}$ starts with thread y. Monadic computations are triggered by a unique thread, whereas functional evaluations may be triggered by more than one thread.*

Example 5.6. We illustrate Definition 5.5 by an example. Let P be the process

$$y_1 \overset{\mathsf{main}}{\Longleftarrow} \texttt{putMVar}\ x_1\ \texttt{True} \mid y_2 \Leftarrow z\ \texttt{True} \mid y_3 \Leftarrow z\ \texttt{False} \mid z{=}(\lambda w_1.\lambda w_2.\texttt{return}\ w_2)\ \texttt{True} \mid x_1\,\mathbf{m} -$$

Then there are two standard reductions for P, i.e. $P \xrightarrow{sr} P_i$ for $i = 1, 2$ where:

$$P_1 := y_1 \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ () \mid y_2 \Leftarrow z\ \texttt{True} \mid y_3 \Leftarrow z\ \texttt{False} \mid z{=}(\lambda w_1.\lambda w_2.\texttt{return}\ w_2)\ \texttt{True} \mid x_1\,\mathbf{m}\,\texttt{True}$$

$$P_2 := y_1 \overset{\mathsf{main}}{\Longleftarrow} \texttt{putMVar}\ x_1\ \texttt{True} \mid y_2 \Leftarrow z\ \texttt{True} \mid y_3 \Leftarrow z\ \texttt{False} \mid z{=}\lambda w_2.\texttt{return}\ w_2 \mid w_1{=}\texttt{True} \mid x_1\,\mathbf{m} -$$

The step $P \xrightarrow{sr} P_1$ is triggered by y_1, and $P \xrightarrow{sr} P_2$ is triggered by y_2 and y_3.

Focusing on a single thread, only the reductions (sr,unIO), (sr,pmvar), and (sr,tmvar) can be seen as a communication with other already existing threads in a reduction sequence reaching a successful state. If the last reduction step of a non-main thread in a reduction sequence reaching success is not of this form, then this reduction step is redundant. We will make this more precise:

Definition 5.7. *Let P be a process and Red be a (finite) reduction sequence from P to a successful process. Let one of the following hold for every reduction step S in Red:*

1. *S is triggered by the main thread.*
2. *S is an (sr,unIO), (sr,pmvar), or (sr,tmvar).*

3. *S is triggered by a thread y, and there is a later step in Red also triggered by thread y.*

Then the reduction Red is called thread-normalized.

Example 5.8. We consider the process P from Example 5.6. Then the reduction sequence $P \xrightarrow{sr} P_1$ is thread-normalized, but for instance the reduction sequence $P \xrightarrow{sr} P_2 \xrightarrow{sr} P_3$, for some P_3, is not thread-normalized since the step $P \xrightarrow{sr} P_2$ does not meet the conditions of Definition 5.7. Indeed, $P \xrightarrow{sr} P_1$ is the only reduction sequence for P which is thread-normalized.

Lemma 5.9. *Let $A \subseteq A_{all}$, P be a process, Red be a reduction sequence from P to a successful process. Then there is also a thread-normalized reduction sequence Red' from P to a successful process that is not longer than Red w.r.t. A.*

Corollary 5.10. *Let P be a process and let $A \subseteq A_{all}$. Then the minimal length $srnr_A(P)$ of sequential reductions of P can be determined by minimizing over thread-normalized reductions.*

For our proof technique, the following rearrangement of reduction sequences is very helpful.

Lemma 5.11. *Let $A \subseteq A_{all}$, P be a process, Red be a thread-normalized reduction sequence from P to a successful process. Let $Red = Red_1; \xrightarrow{sr,a}; Red_2; \xrightarrow{sr,b}; Red_3$, where $b \notin \{(pmvar), (tmvar)\}$, and $\xrightarrow{sr,a}; \xrightarrow{sr,b}$ are triggered by the same thread y, and there is no reduction step in Red_2 that is also triggered by y. Then $Red' = Red_1; \xrightarrow{sr,a}; \xrightarrow{sr,b}; Red_2; Red_3$ is a thread-normalized reduction sequence to a successful process with $srnr_A(Red) = srnr_A(Red')$.*

Proof. There are no conflicts, since $b \notin \{(\text{pmvar}), (\text{tmvar})\}$. Hence the reduction $\xrightarrow{sr,b}$ can be shifted to the left. This does not change the number of reductions in the sequence, for any A.

For proving that transformations are improvements w.r.t. sequential reduction sequences, we define the following improvement-property:

Definition 5.12. *A transformation $\xrightarrow{PT}$ on processes has the* improvement-property for reductions *w.r.t. a set of reduction kinds A, if $\xrightarrow{PT}$ is closed w.r.t. PCtxt-contexts and for all $P \xrightarrow{PT} P'$ and if there is a thread-normalized reduction sequence Red of P to a successful process, then there is a reduction sequence Red' of P' to a successful process, such that $srnr_A(Red) \geq srnr_A(Red')$.*

Due to Lemma 5.9 the improvement-property implies that $\xrightarrow{PT}$ is a sequential A-improvement:

Lemma 5.13. *If a correct transformation $\xrightarrow{PT}$ has the improvement-property for reductions w.r.t. a set of reduction kinds A, then $\xrightarrow{PT}$ is a sequential A-improvement.*

5.3 Proofs of Parallel Improvements

We provide results and arguments also for the last column of Table 1, and thus show that the considered transformations are parallel improvements. Since there are no proper conflicts between sr-reductions, we obtain that standard reductions $\notin \{pmvar, tmvar\}$ are also parallel improvements. The proof is almost the same as for sequential reductions.

Theorem 5.14. *The standard reductions a with $a \notin \{pmvar, tmvar\}$ are parallel improvements for every A and N.*

The notions and analyses of sequential reduction sequences can be transferred to the parallel case. A parallel reduction sequence Red is *thread-normalized* if one corresponding sequential reduction sequence Red_{seq} of Red is thread-normalized (this means all sequences). This can be achieved by thread-normalizing the interleaved reduction sequence. In proofs of parallel improvements, this operation does not increase the length of parallel reduction sequences.

Lemma 5.15. *Let A be a set of reduction kinds and N be a number of processors. Let P be a process and Red be a parallel reduction sequence, using at most N processors, from P to a successful process. Then there is also a thread-normalized parallel reduction sequence Red' from P to a successful process with $srnrp_A^N(Red) \geq srnrp_A^N(Red')$.*

Rearranging sequential reduction sequences is also possible for parallel reduction sequences with certain extra restrictions. This rearrangement is crucial in transferring the arguments for sequential reduction sequences to parallel ones.

We write single reductions steps $\xrightarrow{sr,a}$ in a parallel reductions sequences Red as $Red = Red_0 \ \oplus \ \xrightarrow{sr,a}$.

Lemma 5.16. *Let $A \subseteq A_{all}$, and N be the number of processors. Let P be a process and Red be a parallel thread-normalized reduction sequence from P to a successful process. Let $Red = (Red_1 \ \oplus \ \xrightarrow{sr,a}); Red_2; (Red_3 \ \oplus \ \xrightarrow{sr,b}); Red_4$ with at most N processors, where $b \notin \{(pmvar), (tmvar)\}$, the reductions $\xrightarrow{sr,a}; \xrightarrow{sr,b}$ are triggered by the same thread y, Red_1, Red_3 are single parallel reduction steps or empty, and there is no reduction step in Red_2 triggered by y. In addition we assume that $b \notin A$. Then $Red' = (Red_1 \ \oplus \ \xrightarrow{sr,a}); \xrightarrow{sr,b}; Red_2; Red_3; Red_4$ is also a thread-normalized reduction sequence to a successful process with $srnrp_A^N(Red) = srnrp_A^N(Red')$.*

Proof. Since there are no conflict possibilities (since $b \notin \{(\text{pmvar}), (\text{tmvar})\}$) the reduction $\xrightarrow{sr,b}$ can be shifted to the left. Since $b \notin A$, this rearrangement does not change $srnrp_A^N$.

Note that it is not correct to shift the reduction to the right, due to potential conflicts. Note also that shifting reductions $b \in A$ but $b \notin \{(\text{pmvar}), (\text{tmvar})\}$ can be done, but there is a danger of increasing the number of parallel reduction steps for A, or the number of processors, since Red_3 may contain an A-step.

Lemma 5.17. *Let $A = A_{noncp}$, $Red = Red_{1,1};(Red_{1,2} \oplus \xrightarrow{sr,a_1}); \ldots; Red_{n,1};(Red_{n,2} \oplus \xrightarrow{sr,a_n})$ and N be the number of processors where the following holds:*

1. *$\xrightarrow{sr,a_i}$ are triggered by thread y for all i;*
2. *$Red_{i,1}; Red_{i,2}$ do not contain single reductions triggered by thread y for all i;*
3. *for $i < n$, it holds $a_i \in A_{cp}$,*
4. *$a_n \notin \{(pmvar), (tmvar)\}$*

Then the reduction sequence can be rearranged as $Red' = \xrightarrow{sr,a_1}; \ldots; \xrightarrow{sr,a_{n-1}}; Red_{1,1}; Red_{1,2}; \ldots Red_{n,1}; Red_{n,2} \oplus \xrightarrow{sr,a_n}$ without changing the measure, i.e., $srnrp^N_A(Red) = srnrp^N_A(Red')$.
In the case $a_n \in A_{cp}$, for the shift result $Red' = \xrightarrow{sr,a_1}; \ldots; \xrightarrow{sr,a_n}; Red_{1,1}; Red_{1,2}; \ldots Red_{n,1}; Red_{n,2}$ it holds $srnrp^N_A(Red) = srnrp^N_A(Red')$.

Proof. There are no conflicts in shifting A_{cp}-reductions to the left, and this does not change the $srnrp^N_A$-measure.

Reusing a class of (sequential) forking diagrams can be done as follows:

Lemma 5.18. *A forking diagram with left-down chain $L_1; L_2$ and right-down chain $R_1; R_2$ where $L_1; R_1$ are sequences of A_{cp}-reductions, and L_2, R_2 are at most of length 1, can be applied to a thread-normalized parallel reduction sequence Red, where a thread y is fixed, as follows:*

1. *Rearrange Red such that it is structured as follows: $Red_1; Red_2; Red_3 \oplus r; Red_4$, where Red_1 triggered by thread y is the reduction sequence according to L_1, r (triggered by y) is the reduction according to R_1, where $Red_2; Red_3$ do not contain reduction steps triggered by thread y.*
2. *Replace this by $Red'_1; Red_2; Red_3 \oplus r'_1$, where Red'_1 (triggered by y) are the reductions according to the diagram-labels R_1, and r'_1 (triggered by y) is the reduction step according to R_2.*

Theorem 5.19. *All the transformations treated up to this point for sequential reductions are also improvements for parallel reductions for sets $A \subseteq A_{noncp}$ and any number N of processors. These are all standard reductions with the exception of $\{(sr, pmvar), (sr, tmvar)\}$, and (gc), $(gc)^-$, (ucp), $(ucp)^-$, (cp), (lbeta), (case), (cpcxa), (cpcxb), (mkbinds), (cse), (lunit), (dtmvar), (dpmvar).*

Proof. The arguments for induction are almost the same as for sequential reductions. The differences are that the reduction steps that are not instances of the diagram reductions are the same before and after the transformation.

Looking at all diagrams, we see that these satisfy the preconditions of Lemma 5.18. We explicitly mention all the diagrams where the left down-side has more than one standard reduction:

1. the forking diagram of (ucp) number 3: the left-down reductions are in A_{cp};
2. the third diagram of (ucpt): the given reductions are in A_{cp};
3. the third and 6th diagram of (cse); the first $n-1$ left-down reductions are in A_{cp}, and the last is in $A_{noncp} \setminus \{(pmvar), (tmvar)\}$

```
mainMon = calcMon someTree
calcMon (Leaf n) =
    let res = (g n)
    in  seq res (return res)
calcMon (Node l r) = do
  lres <- (calcMon l)
  rres <- (calcMon r)
  let res  = (lres 'f' rres)
  seq res (return res)
```

```
mainPure' = return (calcPure' someTree)
calcPure' (Leaf n) =  (g n)
calcPure' (Node l r) =
  let
     lres = (calcPure' l)
     rres = (calcPure' r)
     res  = (lres 'f' rres)
  in seq res res
```

Fig. 8. Intermediate Programs for Transforming `mainFut` into `mainPure` from Fig. 1

4. The fourth diagram of lunit: the arguments are the same as for (cse).

Note that there are more diagrams not printed in the paper, but in the supplementary material in the appendix: The fourth forking diagram of (cp); the fifth forking diagram for (cpcxb); and the fifth forking diagram for (mkbinds), which all satisfy the preconditions of Lemma 5.18. Hence the improvement results for sequential reductions also hold for parallel reduction sequences.

Transformation (drfork) which removes the `future`-actions is a sequential improvement whereas the inverse, which parallelizes MVar-access-free actions, is a parallel improvement if the reductions (fork) and (unIO) are not counted. This does not contradict Theorem 5.19, since the arguments are not by a set of forking diagrams, but by an analysis of the rearrangement of the reductions.

6 Conclusion and Further Research

6.1 Applying Improvements

As a part of our conclusion we show how our improvement results enable us to prove detailed properties of concurrent programs. Consider the example program `calcFut` for folding (summing) the values in a tree in the introduction again. We consider the definition of `mainFut` and `mainPure` in Fig. 1 and the programs in Fig. 8. Applying the transformation (drfork) to the `future`-expressions transforms it into `calcMon`. This is justified, since we assume that the tree `someTree` only consists of data. This is a sequential improvement by our results. The transformation `calcMon` to `calcPure'` is a bit more involved, since the recursion has to be restructured. A proof that `mainPure'` is a sequential improvement of `mainMon` can be done by induction on the depth of the tree `someTree` (we omit the details, which are in the appendix in Section C) and under the further assumption of strictness of `f`, i.e. that it requires the value of both arguments, and that values behave like numbers. The comparison of `calcPure'` with `calcPure` shows that `calcPure` is a proper sequential improvement of `calcPure'`, since less (seq), (lunit)-, and (lbeta)-reductions have to be performed, and where other reductions like let-shifting are not counted.

6.2 Related Work on Improvements in Functional Languages

The analysis of improvements for functional languages started with [31] where improvements for call-by-name functional languages are analyzed (for an extension of the lazy lambda calculus see [1] and for the more general lazy computation systems see [13]). An analysis of a non-deterministic call-by-name lambda calculus and an improvement relation based on may- and must-convergence is in [16]. for an untyped call-by-need lambda calculus with `letrec` and data constructors the improvement theory w.r.t. runtime was developed in [20] where also a so-called to tick-algebra was introduced to algebraically compute with improvement laws. A similar investigation, focusing on a formal proof that common subexpression is an improvement w.r.t. runtime, and for a higher-order functional language including Haskell's `seq`-operator, was done in [33] and for a typed variant of the language in [35]. Proof techniques and specific improvement laws for list-processing functions are presented in [34] and [30]. A theory of improvements was developed for a class of languages with structured operational semantics in [32]. Hackett and Hutton [11] used the improvement theory of [20] to argue that optimizations are indeed improvements, with a particular focus on worker/wrapper transformations. This work was extended with a focus on a result which is independent of a concrete programming language (using categorical notions) in [12]. Improvements w.r.t. the space behavior of programs of call-by-need functional programming languages are analyzed by Gustavsson and Sands in [9, 10]. Analyzing space-improvements was recently revived in [5, 6] where a system is presented that supports to measure and analyze the space behavior of typed functional programs w.r.t. call-by-need evaluation.

6.3 Summary and Further Research

We motivated and applied two resource models to the process calculus CHF^* which is a core language for Concurrent Haskell extended by concurrent futures. While sequential improvements consider the interleaved (but sequential) evaluation of concurrent threads, parallel improvements allow parallel execution of concurrent threads. We demonstrated how to show that a specific program transformation is a sequential- and/or a parallel improvement. We proved for a considerable set of transformations that these are sequential and/or a parallel improvements, i.e., optimizations of work and the time of a concurrent evaluation, even for any fixed number of processors. We expect that our optimization methods could also be used in automated and semi-/automated tools for program optimization of Haskell-programs like the tool HERMIT [15, 37]. Our tools, methods and proofs could also be applied if only correctness w.r.t. $\sim_{c,\downarrow}$ is required, i.e., without the precondition of (full) correctness w.r.t. $\sim_c$. We look forward to see applications in these directions.

Future research is proving further and also more complex transformations to be improvements, and to invent and explore further transformation and proof methods. Also a cross-analysis of other resources like space-usage is left for further work. It is also worth studying parallelizations and optimizations during runtime.

References

1. Abramsky, S.: The lazy lambda calculus. In: Research topics in functional programming. pp. 65–116. Addison-Wesley (1990)
2. Ariola, Z.M., Felleisen, M., Maraist, J., Odersky, M., Wadler, P.: A call-by-need lambda calculus. In: POPL'95. pp. 233–246. ACM Press, San Francisco, CA (1995)
3. Baker-Finch, C., King, D.J., Trinder, P.: An operational semantics for parallel lazy evaluation. SIGPLAN Not. 35, 162–173 (2000)
4. Concurrent Clean: Homepage (2017), http://clean.cs.ru.nl/
5. Dallmeyer, N., Schmidt-Schauß, M.: An environment for analyzing space optimizations in call-by-need functional languages. In: Cirstea, H., Escobar, S. (eds.) Proc. 3rd WPTE 2016. EPTCS, vol. 235, pp. 78–92. Open Publishing Association (2017)
6. Dallmeyer, N., Schmidt-Schauß, M.: Space improvements and equivalences in a functional core language (2017), in Informal Proceedings of 4th WPTE 2017, eds. D. Sabel and H. Cirstea
7. Flanagan, C., Felleisen, M.: The semantics of future and an application. J. Funct. Programming 9, 1–31 (1999)
8. Glasgow parallel Haskell: Homepage (2017), http://www.macs.hw.ac.uk/ dsg/gph/
9. Gustavsson, J., Sands, D.: A foundation for space-safe transformations of call-by-need programs. ENTCS 26, 69–86 (1999)
10. Gustavsson, J., Sands, D.: Possibilities and limitations of call-by-need space improvement. In: Pierce, B.C. (ed.) Proc. Sixth ACM ICFP 2001. pp. 265–276 (2001)
11. Hackett, J., Hutton, G.: Worker/wrapper/makes it/faster. In: Jeuring, J., Chakravarty, M.M.T. (eds.) Proc. 19th ICFP 2014. pp. 95–107. ACM (2014)
12. Hackett, J., Hutton, G.: Programs for cheap! In: Proc. 30th ACM/IEEE LICS. pp. 115–126. IEEE Computer Society, Washington, DC, USA (2015)
13. Howe, D.J.: Equality in lazy computation systems. In: Proc. Fourth LICS 1989. pp. 198–203. IEEE Computer Society (1989)
14. Hughes, J.: Why functional programming matters. Comput. J. 32(2), 98–107 (1989)
15. Kansas-University, F.P.G.A.: The Haskell equational reasoning model-to-implementation tunnel (hermit) (2017), http://ku-fpg.github.io/software/hermit/
16. Lassen, S.B., Moran, A.: Unique fixed point induction for McCarthy's Amb. In: Kutylowski, M., Pacholski, L., Wierzbicki, T. (eds.) Proc. 24th MFCS'99. LNCS, vol. 1672, pp. 198–208. Springer (1999)
17. Loogen, R., Ortega-Mallén, Y., Peña Marí, R.: Parallel functional programming in Eden. J. Funct. Programming 15, 431–475 (2005)
18. Marlow, S., Newton, R., Jones, S.L.P.: A monad for deterministic parallelism. In: Claessen, K. (ed.) Proc. 4th Haskell Symposium 2011. pp. 71–82. ACM (2011)
19. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I & II. Inform. and Comput. 100(1), 1–77 (1992)
20. Moran, A.K.D., Sands, D.: Improvement in a lazy context: An operational theory for call-by-need. In: Proc. 26th ACM POPL 1999). pp. 43–56. ACM Press (1999)
21. Niehren, J., Sabel, D., Schmidt-Schauß, M., Schwinghammer, J.: Observational semantics for a concurrent lambda calculus with reference cells and futures. Electron. Notes Theor. Comput. Sci. 173, 313–337 (2007)
22. Niehren, J., Schwinghammer, J., Smolka, G.: A concurrent lambda calculus with futures. Theoret. Comput. Sci. 364(3), 338–356 (2006)
23. Peyton Jones, S.L.: Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: Engineering theories of software construction. pp. 47–96. IOS-Press (2001)

24. Peyton Jones, S.L., Gordon, A., Finne, S.: Concurrent Haskell. In: Proc. 23rd ACM POPL 1996. pp. 295–308. ACM (1996)
25. Peyton Jones, S.L., Singh, S.: A tutorial on parallel and concurrent programming in Haskell. In: Advanced Functional Programming, 6th International School, Revised Lectures. Lecture Notes in Comput. Sci., vol. 5832, pp. 267–305. Springer (2009)
26. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Proc. 20th ACM POPL 1993. pp. 71–84. ACM (1993)
27. Sabel, D.: An abstract machine for concurrent Haskell with futures. In: Jähnichen, S., Rumpe, B., Schlingloff, H. (eds.) Software Engineering 2012 - Workshopband, Fachtagung des GI-Fachbereichs Softwaretechnik, 27. Februar - 2. März 2012 in Berlin. LNI, vol. 199, pp. 29–44. GI (2012)
28. Sabel, D., Schmidt-Schauß, M.: A contextual semantics for concurrent Haskell with futures. In: Schneider-Kamp, P., Hanus, M. (eds.) Proc. 13th ACM PPDP 2011. pp. 101–112. ACM (2011)
29. Sabel, D., Schmidt-Schauß, M.: Conservative concurrency in Haskell. In: Dershowitz, N. (ed.) Proc. 27th IEEE LICS 2012. pp. 561–570. IEEE (2012)
30. Sabel, D., Schmidt-Schauß, M.: A call-by-need lambda calculus with scoped work decorations. In: Zimmermann, W., et.al. (eds.) Software Engineering Workshops 2016. CEUR Workshop Proceedings, vol. 1559, pp. 70–90. CEUR-WS.org (2016)
31. Sands, D.: Operational theories of improvement in functional languages (extended abstract). In: Heldal, R., Holst, C.K., Wadler, P. (eds.) Proc. 1991 Glasgow Workshop on Functional Programming. pp. 298–311. Workshops in Computing, Springer (1991)
32. Sands, D.: From SOS rules to proof principles: An operational metatheory for functional languages. In: Lee, P., Henglein, F., Jones, N.D. (eds.) Proc. 24th ACM POPL 1997. pp. 428–441. ACM Press (1997)
33. Schmidt-Schauß, M., Sabel, D.: Improvements in a functional core language with call-by-need operational semantics. In: Falaschi, M., Albert, E. (eds.) Proc. 7th PPDP 2015. pp. 220–231. ACM (July 2015)
34. Schmidt-Schauß, M., Sabel, D.: Sharing-aware improvements in a call-by-need functional core language. In: Lämmel, R. (ed.) Proc. 27th IFL 2015. pp. 6:1–6:12. ACM, New York, NY, USA (2015)
35. Schmidt-Schauß, M., Sabel, D.: Improvements in a call-by-need functional core language: Common subexpression elimination and resource preserving translations. Science of Computer Programming 147, 3–26 (2017)
36. Schmidt-Schauß, M., Schütz, M., Sabel, D.: Safety of Nöcker's strictness analysis. J. Funct. Programming 18(04), 503–551 (2008)
37. Sculthorpe, N., Farmer, A., Gill, A.: The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In: Proc. 24th IFL 2013. LNCS, vol. 8241, pp. 86–103. Springer (2013)
38. Sestoft, P.: Deriving a lazy abstract machine. J. Funct. Programming 7(3), 231–264 (1997)
39. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.L.: Algorithm + Strategy = Parallelism. J. Funct. Programming 8(1), 23–60 (1998)
40. Wadler, P.: Monads for functional programming. In: First International Spring School on Advanced Functional Programming Techniques. LNCS, vol. 925, pp. 24–52. Springer (1995)

A Equivalence of *CHF* and *CHF**

Theorem A.1. *The calculi CHF and CHF* are equivalent w.r.t. may- and should-convergence, and also w.r.t. correctness of transformations.*

Proof. Let transformation (cpx) be defined as $\mathbb{T}[x] \mid x = y \to \mathbb{T}[y] \mid x = y$ where we assume that it is closed w.r.t. $\mathbb{D}$-contexts and $\equiv$. It suffices to show that $P \downarrow_{CHF^*} \iff P \downarrow_{CHF}$ and $P \Downarrow_{CHF^*} \iff P \Downarrow_{CHF}$ (or equivalently $P \uparrow_{CHF} \iff P \uparrow_{CHF^*}$) for all processes P. Thus we have to show four implications:

1. $P \downarrow_{CHF^*} \implies P \downarrow_{CHF}$: Let $P \xrightarrow{CHF^*,*} Q$, where Q is successful. Then the reduction can be translated into $P \xrightarrow{CHF} \xleftarrow{cpx,*} \xleftarrow{gc,*} P_2 \xrightarrow{CHF} \xleftarrow{cpx,*} \xleftarrow{gc,*} \ldots Q$. Since the reductions (cpx) and (gc) are correct in CHF [28, 29], it is easy to show by induction on the number of $\xrightarrow{CHF}$-reductions, that $P \downarrow_{CHF}$.
2. $P \downarrow_{CHF} \implies P \downarrow_{CHF^*}$: Let $P \xrightarrow{sr,*} Q$, where Q is successful. We transform it into a mixture of reductions and transformations in *CHF**. All sr-reductions are the same, with the exception of (cpcx) which is $(cpcxa);(cpcxb)$ plus equivalences using $\xleftarrow{cpx,*}$ and $\xleftarrow{gc,*}$. The reduction $P_1[x] \mid x = c\ y_1 \ldots y_n \xrightarrow{cpcx} P_1[c\ z_1 \ldots z_n] \mid x = c\ z_1 \ldots z_n \mid z_1 = y_1 \mid \ldots \mid z_n = y_n$ is translated into

$$\begin{array}{l} \xrightarrow{cpcxb} P_1[c\ y_1 \ldots y_n] \mid x = c\ y_1 \ldots y_n \\ \xleftarrow{gc,*} P_1[c\ y_1 \ldots y_n] \mid x = c\ y_1 \ldots y_n \mid z_1{=}y_1 \mid \ldots \mid z_n{=}y_n \\ \xleftarrow{cpx,*} P_1[c\ z_1 \ldots z_n] \mid x = c\ z_1 \ldots z_n \mid z_1{=}y_1 \mid \ldots \mid z_n{=}y_n. \end{array}$$

   where we omit ν-binders. A step-wise transformation of the reduction sequence with the same intermediate processes is of the form $P \xrightarrow{CHF^*} \xleftarrow{gc,*} \xleftarrow{cpx,*} P_2 \xrightarrow{CHF^*} \xleftarrow{gc,*} \xleftarrow{cpx,*} \ldots Q$. This reasoning is also applicable to (cpcxa)-reductions that only abstract some subexpressions. We modify the sequence into a *CHF**-reduction sequence to a successful process: Scanning all possibilities of interference with the sr-reductions of *CHF** are:

$$\begin{array}{ccc} P \xrightarrow{cpx} P' & P \xrightarrow{\quad cpx \quad} P' & P \xrightarrow{cpx} P' \quad P \xrightarrow{gc} P' \\ {\scriptstyle sr,a}\downarrow \quad \downarrow{\scriptstyle sr,a} & {\scriptstyle sr,cpcxb}\downarrow \quad \downarrow{\scriptstyle sr,cpcxb} & {\scriptstyle sr,cp}\downarrow \quad {\scriptstyle sr,a}\downarrow \quad \downarrow{\scriptstyle sr,a} \\ P_1 \xrightarrow[cpx,\mathbb{T}]{} P_1' & P_1 \xrightarrow[cpx]{} \cdot \xrightarrow[cpx,\mathbb{T}]{} P_1' & P_1 \qquad P_1 \xrightarrow[gc]{} P_1' \end{array}$$

   We use these diagrams to shift (gc) and (cpx) to the right, only over *CHF**-reductions. We start with the rightmost of (cpx),(gc). This may increase the cpx-reductions, or it may also remove a cp-reduction using the third diagram. Finally, it leads to a sequence $P \xrightarrow{CHF^*,*} Q'$ $(\xleftarrow{gc,*} \cdot \xleftarrow{cpx,*})^*$ Q. This shifting terminates since the number of *CHF**-reductions is not increased. It is easy to see that also Q' must be successful, since (cpx) and (gc) do no change this property. Hence we have shown that $P \downarrow_{CHF^*}$.
3. $P\uparrow_{CHF^*} \implies P\uparrow_{CHF}$: Analogous to part (1), where Q is CHF*-must-diverging, which is CHF-must-diverging, since part (2) implies $Q\Uparrow_{CHF} \implies Q\Uparrow_{CHF^*}$.
4. $P\uparrow_{CHF} \implies P\uparrow_{CHF^*}$: Let P be a process with a reduction sequence $P \xrightarrow{CHF,*} Q$, where $Q \Uparrow_{CHF}$. We use the same transformation as in part (2), which

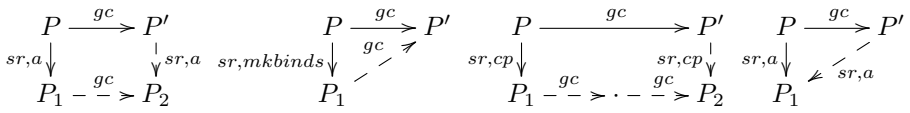
leads to a mixed reduction and transformation sequence $P\ (\xrightarrow{CHF^*} \cdot \xleftarrow{cpx,*} \cdot \xleftarrow{gc,*})^*\ Q$. The diagrams and the shifting process is the same as in part (2), and leads to a sequence $P\ \xrightarrow{CHF^*,*}\ Q'\ (\xleftarrow{cpx,*} \cdot \xleftarrow{gc,*})^*\ Q$. Now we have to argue that also Q' is CHF^*-must-divergent. Since Q is CHF-must-divergent, and since (cpx), (gc) are correct, we also obtain that Q' is CHF-must-divergent, and part (1) implies $Q' \Downarrow_{CHF} \Longrightarrow Q' \Downarrow_{CHF^*}$ and thus Q' is also CHF^*-must-divergent. □

B Detailed Improvement Proofs

B.1 Garbage Collection

Proposition B.1. *(gc) is a sequential A-improvement w.r.t. all sets A.*

Proof. Transformation (gc) is correct (Theorem 5.2). We show that (gc) has the improvement-property for all reduction kinds: Let P be a process with $P{\downarrow}$ and let A be a set of reduction kinds. We obtain the following complete set of forking diagrams for overlaps of (gc) against standard reduction sequences by scanning all cases:



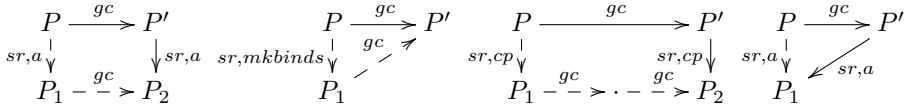
The second case occurs when the whole letrec environment is garbage, the third case occurs, when the redex of the (gc) is within an abstraction, and the fourth case occurs, when the transformation takes place for example in an alternative of a case-expression. In order to show the lemma, we use induction on the number μ of all reduction sequences to show that (i) the number of all reduction steps is not increased, and (ii) that for every reduction kind, the number of reductions is not increased by (gc). The base case is that P is already successful. Then P' is also successful and the claim holds. For the induction step we apply one of the diagrams and the induction hypothesis. The only non-standard case is the third diagram, where we can apply the induction hypothesis twice. □

Now we analyze the inverse reduction of (gc), denoted as $(gc)^-$. Instead of forking diagrams for (gc) we write the diagrams as commuting diagrams for (gc).

Proposition B.2. *Transformation $(gc)^-$ is a sequential A-improvement for all A such that $(mkbinds) \notin A$.*

Proof. Since the transformation (gc) is correct, also $(gc)^-$ is correct. We show that $(gc)^-$ has the improvement property w.r.t. all A with $(mkbinds) \notin A$. Let P be a process with $P{\downarrow}$ and let A be a set of reduction kinds with $(mkbinds) \notin A$.

We obtain the following complete set of forking diagrams for overlaps of $(gc)^-$ against standard reduction sequences by scanning all cases:



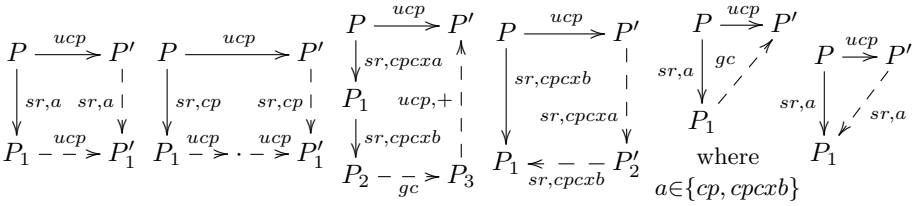
We allow the (exceptional) second diagram which acts like a repeater for a reduction step. This is no problem, since the number of successive $\xrightarrow{sr,mkbinds}$ is finite. In order to show that $(gc)^-$ has the improvement property *w.r.t.* reduction kinds A for A with $(mkbinds) \notin A$, we use induction for the pair (P, P') and the chosen reduction sequence $Red(P')$, and the measure $\mu = (\mu_1, \mu_2, \mu_3)$, where μ_1 is the number of all reductions kinds with the exception of (mkbinds) of $Red(P')$, μ_2 is the number of all reduction kinds in $Red(P')$, and μ_3 is the number of letrec-symbols in P'. The claim is that the number of all reductions $\neq (mkbinds)$ is not increased by $(gc)^-$. If there is no reduction sequence, then we see that (gc) does not change successfulness. For diagram 1, the induction hypothesis is applicable. For diagram 2, μ_1, μ_2 are the same, but μ_3 is strictly decreased, and we can apply the induction hypothesis. For the third diagram, we apply the induction hypothesis twice. For the fourth diagram, the conclusion is immediate. □

B.2 Unique Copying

We present the proof for (ucp), since inlining is an often used transformation also used in other proofs. The transformation (ucp) may increase the number of (cpcxa)-steps in a reduction sequence, since it is the immediate inverse in certain cases.

Proposition B.3. *(ucp) is a sequential A-improvement for all A s.t. $(cpcxa) \notin A$.*

Proof. Transformation (ucp) is correct (Theorem 5.2). We show that (ucp) has the improvement-property for reduction sequences for all sets A of reduction kinds with $(cpcxa) \notin A$. Let P be a process with $P{\downarrow}$ and let $(cpcxa) \notin A$. A complete set of forking diagrams for thread-normalized reduction sequences is:



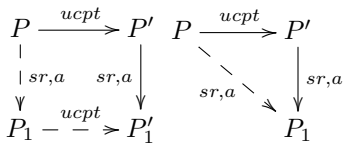
Note that the third diagram can only be used for thread-normalized reduction sequences, where the left down-arrows are for a common thread. Let $P \xrightarrow{ucp} P'$ and let *Red* be a thread-normalized reduction sequence of P to a successful

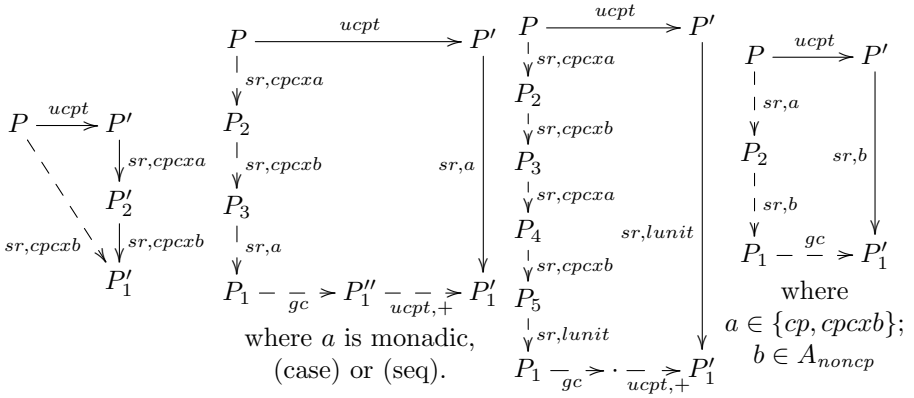
process, where $P \xrightarrow{sr,a} P_1$ is the first reduction of *Red*. We use $\mu = (\mu_1, \mu_2)$ as measure for induction, where μ_1 is the number of all reductions with the exception of (cpcxa), and μ_2 is the number of all reductions, and the pair is ordered lexicographically. We show two claims: (i) that there is a reduction sequence Red' of P' such that $\mu_1(Red') \leq \mu_1(Red)$; and (ii) that for every reduction kind $a \neq (cpcxa)$ its number in the reduction sequence is decreased. If $\mu_1 = 0$, then P is successful and then also P' is successful. If $\mu_1 > 0$, then we apply a forking diagram. If the first diagram is applicable, then we can apply the induction hypothesis to P_1. In the cases of the second diagram, we can apply the induction hypothesis twice, since the first claim holds, and then obtain the two claims. In the case of the third diagram, since *Red* is thread-normalized, the reduction (cpcxa) cannot be the last one for all threads y triggering it, hence a (cpcxb) must be a later reduction for some of these threads. Proposition B.1 shows that we can apply the induction hypothesis to P_3, and then several times until P', which shows that first claim. The diagrams and Proposition B.1 then also show the second claim. For the 4^{th} diagram, we obtain immediately that the reduction sequence for P' has not more reduction steps $\neq$ cpcxa, and the second claim on the number of occurrences of every reduction kind holds. For the 5^{th} diagram, Proposition B.1 can be applied and shows the claim. Also the second claim holds. For the 6^{th} diagram, the induction hypothesis can be applied, and then the two claims hold. □

We treat the inverse of (ucp), denoted as $(ucp)^-$ and first consider the inverse $(ucpt)^-$of (ucpt), and thereafter we consider the inverse $(ucpd)^-$ of (ucpd).

Proposition B.4. *$(ucpt)^-$ is a sequential A-improvement for all $A \subseteq A_{noncp}$.*

Proof. It suffices to show that $(ucpt)^-$ has the improvement-property for reduction sequences w.r.t. all $A \subseteq A_{noncp}$, since $(ucpt)^-$ is correct. A complete set of forking diagrams for $(ucpt)^-$ is:

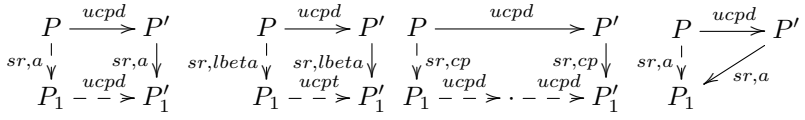




Let $P \xrightarrow{ucpt} P'$ and let $P'\!\downarrow$ and let Red' be a thread-normalized reduction sequence to a successful process from P'. We show two claims: (i) there is a reduction sequence Red of P such that $\mu_1(Red) \leq \mu_1(Red)$; where μ_1 is the number of all reductions in A_{noncp}; (ii) for every reduction kind $a \notin A_{noncp}$ its number in the reduction sequence is decreased. We use $\mu = (\mu_1, \mu_2)$ as measure for induction, where μ_2 is the number of all reduction steps, and the pair is ordered lexicographically. If P' is successful, then P is also successful, or a single (cpcxb)-step makes P successful, hence the claim holds. Otherwise, let $P' \xrightarrow{sr,a} P'_1$ be the first reduction of Red'. If the first diagram is applicable, then we apply the induction hypothesis to P'_1. For the second diagram, the claim is trivial. For the third diagram, we can assume that Red' is thread-normalized, and counting the number of reductions shows the claim. For the fourth diagram, the induction hypothesis is applied to P'_1 and then to the right until the claim holds for P''_1, but with a strictly smaller μ_1 than for P'. Since (gc) is an improvement-equivalence, P_1 has a standard reduction sequence with a smaller μ_1-measure than $\mu_1(P'_1)$, and we can count μ_1 for the constructed reduction sequence of P, which shows the claim. The same arguments can be applied to diagram 5. For the 6^{th} diagram, similar arguments show the claim. □

Proposition B.5. *$(ucpd)^-$ is a sequential A-improvement for all $A \subseteq A_{noncp}$.*

Proof. Correctness of $(ucpd)^-$ holds by Theorem 5.2. To show the improvement-property for reduction sequences for all A with $A \subseteq A_{noncp}$, let P be a process with $P\!\downarrow$ and let $A \subseteq A_{noncp}$. Since (ucpd) is applied only in abstractions, the following is a complete set of forking diagrams:



Now let $P \xrightarrow{ucpd} P'$. P is successful iff P' is, and in this case the claim holds. If P is not successful, then let Red' be a reduction sequence of P' to a successful

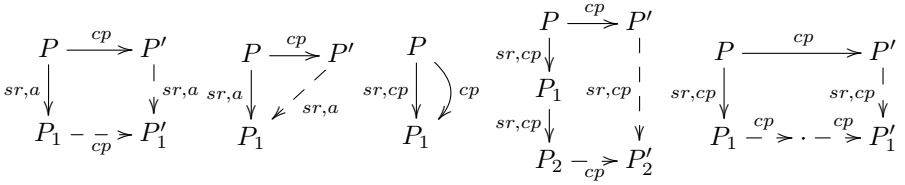
process, where $P' \xrightarrow{sr,a} P'_1$ is the first reduction of Red'. We use $\mu = (\mu_1, \mu_2)$ as measure for induction, where μ_1 is the number of (lbeta)-reductions, and μ_2 is the number of non-(lbeta)-reductions before the first (lbeta)-reduction. and the pair is ordered lexicographically. We show two claims: (i) that there is a reduction sequence Red of P such that $\mu(Red) \leq \mu(Red')$; and (ii) that for every set $A \subseteq A_{noncp}$ the length w.r.t. A remains the same, i.e., $srnr_A(Red) = srnr_A(Red')$. First we show claim (i). If the first diagram is applicable, then the induction hypothesis is applicable, and the claim can be shown. If the second diagram is applicable, then Proposition B.4 can be applied for the reduction (lbeta) which shows claim (i). If the third diagram applies, then we can apply the induction hypothesis twice, and derive reduction sequences Red''_1 and Red_1 with $\mu(Red_1) \leq \mu(Red''_1) \leq \mu(Red'_1)$, and then claim (i) can be shown. For the 4^{th} diagram, the claim 1 is obvious. Now we can show claim (ii) for reduction kinds $A \subseteq A_{nocp}$ by induction on the measure μ using the already proved claim (i) and Proposition B.4. □

B.3 The Transformations (cp), (cpcxa), (cpcxb), (lbeta), (case), (seq) and (mkbinds)

We consider the copy transformation (cp) and argue that it a sequential improvement using the tool of forking diagrams.

Proposition B.6. *Transformation (cp) is a sequential A-improvement for all reductions kinds A.*

Proof. The transformation (cp) is correct (Theorem 5.2) and thus it suffices to show that (cp) has the improvement-property for reductions for all sets A of reductions kinds. Let P be an expression with $P{\downarrow}$, $P \xrightarrow{cp} P'$, and let Red be a successful reduction for P. We show that there is a successful reduction for P' that is not longer than Red w.r.t. A: The case that P is successful is trivial. So let us assume that P' is not successful. A complete set of forking diagrams is:



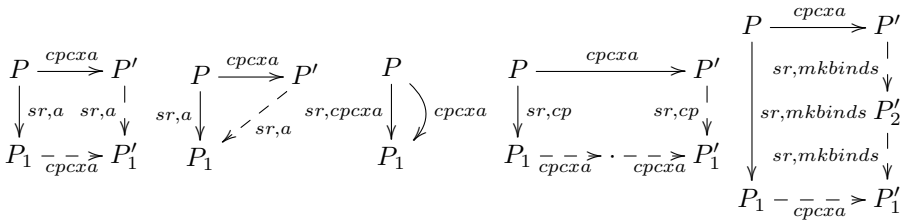
The forking diagrams permit us to apply the induction hypothesis to the reduction sequence Red of P, where we use the number of all reductions as measure, and the claims are: (i) that the number of all reductions is not increased, and that for every reduction kind, its number of occurrences is not increased. In the case of the diagram 5 we have to apply the induction hypothesis twice, and can then show that the total number of reductions of P' is not greater. In case of diagram 4, we replace Red by a thread-normalized reduction sequence, which is possible due to Corollary 5.10, and can commute the reduction sequence such

that the diagram is applicable. This shows also that the minimal length of reductions of P' is not larger than that of P. □

The difference between (cpcxb) and (cp) is that (cpcxb) has (cpcxa) as an inverse in special cases. Transformation (cpcxb) may increase the total number of reductions by a number of (cpcxa)-reductions (see diagram 3 below), hence the forking diagrams must be more detailed.

Proposition B.7. *The transformation (cpcxa) is a sequential A-improvement for all sets A of reductions kinds with* $(mkbinds) \notin A$.

Proof. The transformation (cpcxa) is correct (Theorem 5.2) and thus it suffices to show that (cpcxa) has the improvement-property for reductions w.r.t. all A s.t. $(mkbinds) \notin A$. Let A be a set of reduction kinds with $(mkbinds) \notin A$. Let P be an expression with $P{\downarrow}$, $P \xrightarrow{cpcxa} P'$, and let *Red* be a successful reduction for P. We show that there is a successful reduction for P' that is not longer than *Red* w.r.t. A. The cases that P or P' are successful is trivial. So let us assume that P, P' are not successful. A complete set of forking diagrams for (cpcxa) is:



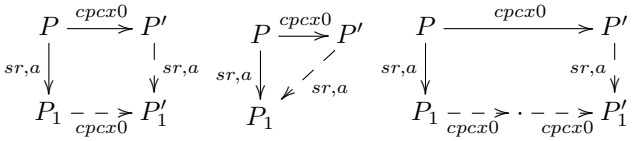
The induction proof is done using the measure is $\mu = (\mu_1, \mu_2)$, where μ_1 is the number of all reductions $\neq mkbinds$, and μ_2 is the number of all reductions. The claims are: (i) μ_1 is not increased by (cpcxa); and (ii) the number of reduction kinds in A is not increased. The first claim is easy for diagrams 1,2,3. For the 4^{th} diagram, the induction hypothesis can be applied twice, since μ_1 is not increased. For diagram 5, the induction hypothesis is applicable. The second claim follows by an easy induction using the same measure. □

The transformation (cpcx0) is used in the improvement proof of (cpcxb). It is defined as follows where c is a constructor or monadic operator:

$$\mathbb{T}[y] \mid y = c\ x_1 \ldots x_n \mid u = c\ x_1 \ldots x_n \quad \rightarrow \quad \mathbb{T}[u] \mid y = c\ x_1 \ldots x_n \mid u = c\ x_1 \ldots x_n$$

Proposition B.8. *(cpcx0) is a sequential A-improvement for all sets A.*

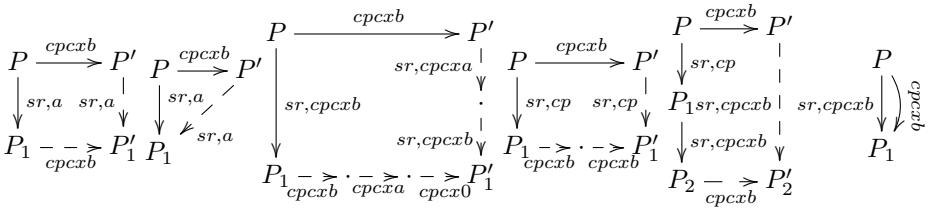
Proof. Correctness of (cpcx0) follows from [28, 29] and Theorem A.1. We show that (cpcx0) has the improvement-property for reduction sequences w.r.t. all A. Let $P{\downarrow}$, $P \xrightarrow{cpcx0} P'$, and *Red* be a successful reduction sequence for P. We show that there is a successful reduction sequence for P' that is not longer than *Red* w.r.t. A: The cases that P or P' are successful are trivial. Assume that P, P' are not successful. A complete set of forking diagrams for (cpcx0) is as follows:



The induction for the claim that there is a successful reduction sequence of P' with the same number of A-reductions is straightforward using the number of all reduction steps as measure. This shows the claim on the improvement property. □

Proposition B.9. *The transformation (cpcxb) is an improvement for every set A of reduction kinds with $\{(mkbinds), (cpcxa)\} \cap A = \emptyset$.*

Proof. The transformation (cpcxb) is correct (Theorem 5.2) and thus it suffices to show that (cpcxb) has the improvement-property for reduction sequences w.r.t.. all A s.t. $\{(mkbinds), (cpcxa)\} \cap A = \emptyset$. Let P be an expression with $P{\downarrow}$, $P \xrightarrow{cpcxb} P'$, and let *Red* be a successful reduction sequence for P. We show that there is a successful reduction sequence for P' that is not longer than *Red* w.r.t. A. The case that P or P' are successful is trivial. So let us assume that P is not successful. A complete set of forking diagrams for (cpcxb) is as follows:



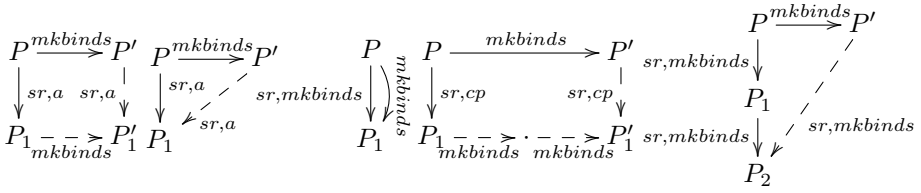
We assume that the reduction sequence of P is thread-normalized. The induction measure is $\mu = (\mu_1, \mu_2)$ where μ_1 is the number of reduction steps not of reduction kind (cpcxa) or (mkbinds). and μ_2 is the number all reductions. The claims are: (i) that the measure μ_1 is not increased, and (ii) that for every reduction kind $a \notin \{(cpcxb), (mkbinds)\}$, the length of reduction sequences is decreased w.r.t. a.

This can be shown using the diagrams for (cpcxb): In case of the first diagram, we apply the induction hypothesis to P_1. For the second diagram, the proof is immediate. For the 3rd diagram, the induction hypothesis can be applied to P_1. Using Propositions B.7 and B.8, we see that that $\mu_1(Red(P_1')) \leq \mu_1(Red(P_1))$, hence $\mu_1(Red(P')) \leq \mu_1(Red(P))$, where $Red(..)$ means the given or constructed reduction sequence. Note that (mkbinds) have to be excluded from the counting, since (cpcxa) may increase the number of (mkbinds). For the $4th$ diagram, the induction hypothesis can be applied twice, since $cp \notin \{cpcxa, mkbinds\}$. For the 5^{th} diagram, the induction hypothesis can be applied. Here we have exploit that the reduction sequence can be thread-normalized and rearranged without making it longer. For the 6^{th} diagram, the proof is immediate.

The second claim is proved easily. Thus, the improvement property follows, since we have proved the non-increasing property for all reductions. □

Proposition B.10. *(mkbinds) is a sequential A-improvement for all A.*

Proof. This is similar to previous simple proofs. We display the forking diagrams:



An induction on the number of all reductions shows the claim. □

Treating the transformations (lbeta), (case), and (seq) is straightforward. An induction on the length of a standard reduction sequence, where the forking diagrams are of a shape where the methods can be applied as usual, can be used to show:

Proposition B.11. *(lbeta), (case), and (seq) are sequential A-improvements for all A.*

Proposition B.12. *We summarize the results:*

1. *Transformation (cp) is a sequential A−improvement for all A.*
2. *Transformation (cpcxa) is a sequential A-improvement if* $(mkbinds) \notin A$.
3. *The program transformation (cpcxb) is a sequential A-improvement for all A with* $A \subseteq A_{all} \setminus \{(mkbinds), (cpcxa)\}$.
4. *(lbeta), (case), and (seq) are sequential A-improvements for all A.*

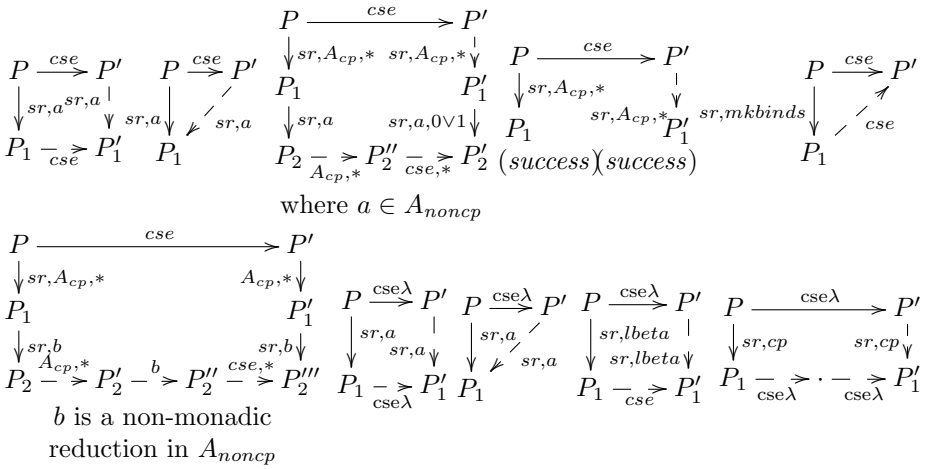
B.4 Common Subexpression Elimination

We write $\xrightarrow{A_{cp}}$ and $\xrightarrow{A_{cp},*}$ for a reduction or a sequence, respectively, where the reduction kinds are in A_{cp}. Correspondingly, we use $\xrightarrow{A_{noncp}}$ and $\xrightarrow{A_{noncp},*}$.

Proposition B.13. *(cse) is a sequential A-improvement for all* $A \subseteq A_{noncp}$.

Proof. Theorem 5.2 shows that (cse) is correct. For proving that (cse) has the improvement-property for reductions w.r.t. all A with $A \subseteq A_{noncp}$, let $P \xrightarrow{cse} P'$, *Red* be a successful reduction for P, and $A \subseteq A_{noncp}$. We show that there is a successful reduction *Red*′ for P' that is not longer w.r.t. A than *Red*: The case that P is successful is trivial. Assume that P is not successful. We compute a complete set of forking diagrams, where we distinguish between (cse) applied at

positions that are not in a body of an abstraction, and (cse) applied within a body of an abstraction as (cseλ).



First we prove the improvement property for (cse) that is not applied in abstractions. The proof of the improvement property is by induction on the length $\mu(Red) = (\mu_1(Red), \mu_2(Red))$ of a reduction Red, $\mu_1(Red) = srnr_{A_{noncp}}(Red)$ and $\mu_2(Red)$ is the number of all reductions. Let P, P' be processes with $P{\downarrow}$ and $P \xrightarrow{cse} P'$. If P is successful, and P' is not, then $\xrightarrow{cse}$ is an application of (cse) immediately to the top expression in the expression of the main thread. At most two standard reductions $\xrightarrow{sr,A_{cp}}$ are sufficient to again get a successful process.

If P is not successful, then we fix a thread-normalized reduction Red to a successful process. It suffices to look at diagrams 3 and 4, 5, 6 which cover all cases. If Red does not contain a reduction from A_{noncp},then thread-normalization and rearrangement of the reduction permit to apply diagram 4, which shows that there is a reduction from P'. In the other case Red is a thread-normalized reduction that contains a reduction step from A_{noncp}. After applying a rearrangement, diagrams 3, 5, or 6 can be applied: In case of diagram 3, the induction hypothesis can be applied to P_2. The diagram shows that there is a transformation sequence $P_2 \xrightarrow{A_{cp},*} P_2'' \xrightarrow{cse,*} P_2'$. Then Propositions B.12 and B.3 show that there is a (successful) reduction Red'' of P_2'' with $srnr_{A_{noncp}}(Red'') \leq srnr_{A_{noncp}}(Red)$, hence the induction hypothesis is also applicable to P_2'', and also for the other intermediate processes, which shows that there is a (successful) reduction Red' of P_2' with $srnr_{A_{noncp}}(Red') \leq srnr_{A_{noncp}}(Red'') \leq srnr_{A_{noncp}}(Red)$. In case of diagram 5, the reasoning is easy. In case of diagram 6, reasoning is similar to the case of diagram 4, however, we also need Proposition B.12 for the horizontal $\xrightarrow{b}$-reduction. The missing arguments are similar to the previous ones. This concludes the induction proof for (cse).

Now we prove the property for (cseλ) using the result for (cse) not applied within abstractions. We use induction on $\mu(Red) = (\mu_1(Red), \mu_2(Red))$ of a reduction Red, where $\mu_1(Red) = srnr_{A_{noncp}}(Red)$ and $\mu_2(Red)$ is the number

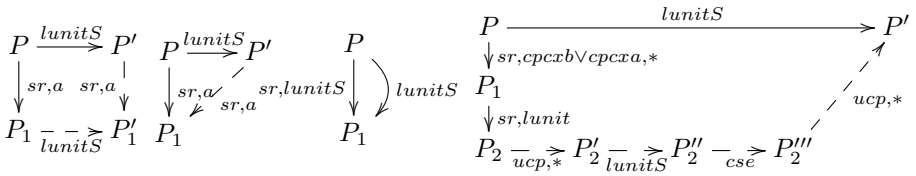
of A_{cp}-reductions before the first noncp-reduction. Here we do not use rearrangement of reductions. The claim is that the measure is not changed by the diagrams. Looking at the diagrams for (cseλ): for 1,2 this can be proved by applying the induction hypothesis. For diagram 3, the previous result for (cse) can be applied, and for diagram 4, the induction hypothesis can be applied twice. □

B.5 Transformation (lunit)

Correctness of (lunit) requires typing, since for instance, the untyped process $P := y \overset{\mathsf{main}}{\Longleftarrow} \mathtt{case_{Bool}}\,((\mathtt{return\,True}) \gg\!= (\lambda x.x))\ (\mathtt{True} \to \mathtt{return\,True})\ldots)$ gets stuck, and is non-converging, while P can be transformed by (lunit) into the process $y \overset{\mathsf{main}}{\Longleftarrow} (\mathtt{case_{Bool}}\,((\lambda x.x)\,\mathtt{True})\,(\mathtt{True} \to (\mathtt{return\,True})\ldots)$, which reduces to the successful process $y \overset{\mathsf{main}}{\Longleftarrow} \mathtt{return\ True}$. We distinguish (lunit) into transformations (lunitS) and (lunitd), where the first is (lunit) applied in surface contexts, and the latter is (lunit) applied within abstractions.

Proposition B.14. *(lunit) is a sequential A-improvement for every $A \subseteq A_{noncp}$*

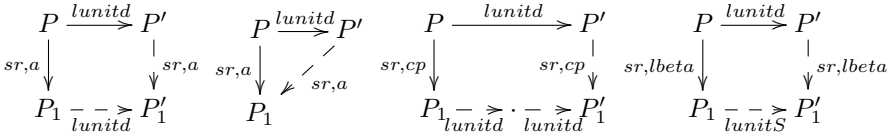
Proof. A complete set of forking diagrams for (lunitS) is:



Let P be an expression with $P \xrightarrow{lunitS} P'$, and let Red be a successful reduction for P. Let $A \subseteq A_{noncp}$. We show that there is a successful reduction Red' for P' that is not longer w.r.t. A than Red: The case that P is successful is trivial. So let us assume that P is not successful. We use the complete set of forking diagrams for (lunit) to make the induction, which is on the following measure of a reduction Red: $\mu(Red) = (\mu_1(Red), \mu_2(Red))$, where μ_1 is $srnr_{A_{noncp}}(Red)$ and μ_2 is $srnr_{A_{all}}(Red)$, and the measure is ordered lexicographically.

The claims are (i) that there is a reduction Red' of P' with $srnr_{A_{noncp}}(Red) \geq srnr_{A_{noncp}}(Red')$ and (ii) that this reduction satisfies $srnr_A(Red) \geq srnr_A(Red')$. We check the diagrams in turn. Diagram 1 permits application of the induction hypothesis to P_1, and the claim is easy. For diagram 2 and 3 the proof is obvious. For diagram 4, Red can be assumed to be thread-normalized. For Red_2 at P_2, the application of Proposition B.3 shows that there is a reduction Red'_2 at P'_2 with $\mu_1(Red'_2) < \mu_1(Red)$. Thus the induction hypothesis can be applied, and we obtain a reduction Red'' of P''_2 with $\mu_1(Red''_2) \leq \mu_1(Red')$. Now propositions B.13 and B.3 show that there is a reduction Red' from P' with $\mu_1(Red') < \mu_1(Red)$. This proves the claim for transformation (lunitS). Now we inspect the

transformation (lunitd). A complete set of forking diagrams for (lunitd) is:



The claim is that for $A \subseteq A_{noncp}$, P with $P{\downarrow}$, and $P \xrightarrow{lunitd} P'$, and a reduction Red of P, there is a reduction Red' of P' such that $srnr_A(Red') \leq srnr_A(Red)$. We use induction with the measure: $\mu(Red) = (\mu_1(Red), \mu_2(Red))$, ordered lexicographically, where μ_1 is is the number of lbeta-reductions in Red and μ_2 is the number of other reductions before the first (lbeta)-reduction.

First we prove that (lunitd) does not increase the measure μ: Scanning the diagrams, we see: For diagram 1 we apply the induction hypothesis. For diagram 2 the reasoning is obvious. For diagram 3, the induction hypothesis can be applied twice; and for diagram 4, we can apply the induction hypothesis and the result above for (lunitS). This shows the first claim.

The main claim can now be proved using the same schema and steps. □

B.6 Proof of Lemma 5.9

Lemma 5.9. *Let $A \subseteq A_{all}$, P be a process, Red be a reduction sequence from P to a successful process. Then there is also a thread-normalized reduction sequence Red' from P to a successful process that is not longer than Red w.r.t. A.*

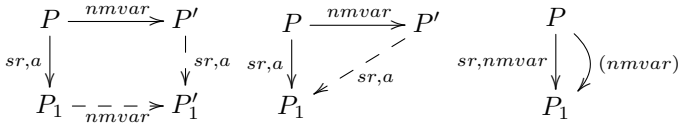
Proof. Let S be the last reduction step in Red among the reduction steps that violate Definition 5.7 of thread-normalized. If S is a monadic computation different from (sr,unIO), (sr,pmvar), or (sr,tmvar), then it is triggered by a single thread y, and the reduction Red' constructed by omitting S also leads to a successful process, since no thread different from y can see the effect of the reduction. If S is a functional computation, then it may be triggered by several threads $y_1, \ldots, y_n$, and there is no later reduction in Red triggered by any of the threads $y_1, \ldots, y_n$. Then again we can construct a reduction sequence Red', where the reduction step S is omitted, and since no thread requires the result of S, the reduction Red' leads to a successful process. □

B.7 More Monadic Transformations

We investigate (nmvar), and the deterministic versions (dtmvar), (dpmvar) of take and put on an MVar, and show that these are improvements.

Proposition B.15. *(nmvar) is a sequential A-improvement for all A.*

Proof. The transformation (nmvar) is correct,. Computing the forking diagrams results only in the trivial diagrams, since there are no conflicts.



Now it is easy to see by induction that for every reduction sequence of a process P, there is also one for process P' where the number of reductions is not increased, for all reduction kinds. □

Proposition B.16. *The transformations (dtmvar) and (dpmvar) are sequential A-improvements for all A.*

Proof. The transformations are correct. Let $P \xrightarrow{dtmvar \vee dpmvar} P'$. Due to the conditions on the transformations, for every reduction sequence of P, there is also one for P' which is the same, but only the (tmvar) or (pmvar) that corresponds to the transformation is omitted. Hence both transformations are improvements. □

It is obvious that (sr,tmvar), (sr,pmvar) in general are not correct. Transformation (sr,fork) is correct and an improvement, however, (fork) as a non-standard reduction is in general not correct, since for instance, the process $\texttt{main} \Leftarrow (\texttt{takeMVar}\ x)\ \texttt{>>=}\ \lambda\_.(\lambda y.\texttt{return True})(\texttt{fork}\ (\texttt{takeMVar}\ x))\ \texttt{|}\ x\ \texttt{m True}$ is should-convergent, but the transformation result $\texttt{main} \Leftarrow (\texttt{takeMVar}\ x)\ \texttt{>>=}\ \lambda\_.(\lambda y.\texttt{return True})(\texttt{return}\ z)\ \texttt{|}\ x\ \texttt{m True}\ \texttt{|}\ z \Leftarrow \texttt{takeMVar}\ x$ is may-divergent: If thread $(z \Leftarrow \texttt{takeMVar}\ x)$ fires first, reduction is blocked. Hence, the (fork) transformation can only be correct and an improvement under further restrictions.

B.8 The Transformation (drfork)

Proposition B.17. *The transformation (drfork) is correct.*

Proof (Sketch). Let $P \xrightarrow{drfork} P'$ and *Red* be a successful reduction sequence of P. Assume that $P = \mathbb{D}[y \Leftarrow \texttt{future}\ e]$, $P' = \mathbb{D}[y \Leftarrow e]$. We make an analysis of the changes due to by (drfork): Let us first assume that *Red* does not contain further (fork)-reductions for e. Then $P \xrightarrow{sr} P_1$, which is of the form $\mathbb{D}[\nu z.(y \Leftarrow \texttt{return}\ z\ \texttt{|}\ z \Leftarrow e)]$. We can assume that *Red* is minimal in the sense that there are no reduction steps that can be erased without changing the property of being successful. The reduction sequence *Red* can be rearranged such that the reduction steps for z are preferred. This holds, since there are no MVar-accesses triggered by thread z due to the assumption on (drfork). The minimality assumption now shows that the reduction sequence for thread z ends with an (unIO). The reduction steps before (unIO) can also be done for P'. Then on the P side we obtain a process P_2, and on the P'-side a process P'_2, such that

$P_2 \xrightarrow{ucp} P_2'$. Since P_2 has a successful reduction sequence, we see that also P_2' has a successful reduction sequence, since (ucp) is correct.

This argument can be extended to more occurrences of (deterministic) (fork) in the reduction sequence. Let *Red* be a successful reduction sequence of P. Again we can assume that *Red* is minimal in the sense that there are no reduction steps that can be erased without changing the property of being successful. Now we modify the reduction sequence *Red* as follows: We start with the reduction steps of a forked (deterministic) thread. Due to the assumption that there are no MVar-accesses triggered by the thread, say z, shifting reduction steps permits to have the reduction steps triggered by z in a contiguous sequence. We add a final (ucp) to remove the created binding for z and inline it again. This can be done for all deterministic threads, where the intermediate reduction sequence may also have interspersed (ucp)-transformations. Finally, the reduction sequence for P' is constructed by working backwards through the reduction sequence: remove (unIO) and (fork) coming from to deterministic threads, and use the correctness and improvement equivalence of (ucp) to create a (ucp)-free reduction sequence for P' to a successful process. An analogous analysis shows that a successful reduction sequence for Red' of P' can be transferred into a reduction sequence *Red* of P, by inserting the necessary (fork)- and (unIO)-reductions and using $(\text{ucp})^-$. This shows that $P{\downarrow} \iff P'{\downarrow}$.

The same analysis can be made for may-divergence in both directions, which shows that P and P' are equivalent w.r.t. may- and should-convergence. □

Proposition B.18. *(drfork) is a sequential A-improvement for all* $A \subseteq A_{noncp}$.

Proof. Let $P \xrightarrow{drfork} P'$ and *Red* be a successful reduction sequence of P. The analysis in the previous proof shows that only reduction steps are removed and $\xrightarrow{ucp}$ is used. Hence there is shorter reduction sequence of P' w.r.t. A_{noncp}. □

Remark B.19. The transformation (drfork) is a not a parallel improvement, since the modification of a parallel reduction sequence by omitting a fork leads to an increase of the length of the reduction sequence.

The inverse transformation of (drfork), which can be seen as a parallelization, is in general not a parallel improvement, since reductions (fork) and (unIO) may be added in reduction sequences. However, there is a good chance that the parallelization may have an advantage over interleaved reduction sequences.

Proposition B.20. *The inverse* $(drfork)^-$ *of the transformation (drfork) is a parallel A-improvement for all* $A \subseteq (A_{noncp} \setminus \{(fork), unIO\})$.

Proof. The analysis of reductions as above shows that (ignoring the reduction kinds A_{noncp}), only (fork) and (unIO) are added. □

C Arguments for Examples

We show the relation between `mainFut`, `mainMon`, `mainMon'`, and `mainPure''`:

```
mainMon = calcMon someTree
calcMon (Leaf n) =
    let res = (g n)
    in  seq res (return res)
calcMon (Node l r) =
 do lres <- (calcMon l)
    rres <- (calcMon r)
    let res  = (lres `f` rres)
    seq res (return res)

mainMon' = calcMon' someTree
calcMon':: Tree -> IO Integer
calcMon' (Leaf n) =
  let res = (g n)
  in  seq res (return res)
calcMon' (Node l r) =
  (calcMon' l)  >>=
   (\lres ->  (calcMon' r)
    >>=
    (\rres -> let
                res =
                 (lres `f` rres)
               in return res))
```

```
mainPure' =
   return (calcPure' someTree)
calcPure' (Leaf n) =  (g n)
calcPure' (Node l r) =
  let
     lres = (calcPure' l)
     rres = (calcPure' r)
     res  = (lres `f` rres)
  in seq res res

mainPure'' :: IO Integer
mainPure'' =
  let res =
       (calcPure'' someTree)
  in seq res (return res)
calcPure''':: Tree -> Integer
calcPure'' (Leaf n) =  (g n)
calcPure'' (Node l r) =
  (\lres ->
    (\rres ->
       (lres `f` rres))
    (calcPure'' r))
  (calcPure'' l)
```

The comparison between `mainMon'` and `mainPure''` results in:

Lemma C.1. *The results of* `mainPure''` *and* `mainMon'` *are identical. The transformation of* `mainMon'` *into* `mainPure''` *is a sequential improvement: it requires two more (lunit)-reductions per node of* `sumTree`.

Proof. An induction proof on the depth of the tree `someTree` shows that the result of `let` res=(`calcPure'' someTree`) `in seq` res (`return` res) is identical to (`calcMon' someTree`). Correctness follows from correctness of (lunitS). The improvement property follows from Proposition B.14. □

Lemma C.2. *The transformation from* `calcPure''` *into* `calcPure'` *is a sequential and parallel improvement:*

Proof. This follows from the improvement property of (lbeta) and the improvement equivalence of the let-transformations. □