A Contextual Semantics for Concurrent Haskell with Futures

David Sabel and Manfred Schmidt-Schauß

Dept. Informatik und Mathematik, Inst. Informatik, J.W. Goethe-University,
PoBox 11 19 32, D-60054 Frankfurt, Germany,
{sabel,schauss}@ki.informatik.uni-frankfurt.de

Technical Report Frank-44

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

March 14, 2011

Abstract. In this paper we analyze the semantics of a higher-order functional language with concurrent threads, monadic IO and synchronizing variables as in Concurrent Haskell. To assure declarativeness of concurrent programming we extend the language by implicit, monadic, and concurrent futures. As semantic model we introduce and analyze the process calculus CHF, which represents a typed core language of Concurrent Haskell extended by concurrent futures. Evaluation in CHF is defined by a small-step reduction relation. Using contextual equivalence based on may- and should-convergence as program equivalence, we show that various transformations preserve program equivalence. We establish a context lemma easing those correctness proofs. An important result is that call-by-need and call-by-name evaluation are equivalent in CHF, since they induce the same program equivalence. Finally we show that the monad laws hold in CHF under mild restrictions on Haskell's seq-operator, which for instance justifies the use of the do-notation.

1 Introduction

Futures are variables whose value is initially not known, but becomes available in the future when the corresponding computation is finished (see e.g. [BH77,Hal85]). For functional programming languages the call-by-need evaluation implements futures (implicitly) on the functional level, since shared expressions are evaluated at the time their value is demanded. Nevertheless in this paper we will consider *concurrent* futures on the *imperative* level in the functional programming language Haskell [Pey03].

The futures presented in this paper are *concurrent*, since the computation necessary to obtain the value of a future is performed in a *concurrent thread*. We consider the *imperative* level, since the value of a future is obtained by performing stateful programming, i.e. it is performed as a monadic computation in Haskell's `IO`-monad (see e.g. [PW93,Wad95,Pey01]).

One also distinguishes between *explicit* futures, i.e. where the value of a future must be explicitly forced and *implicit* futures where the value is computed automatically if the value is demanded by data dependency, i.e. there is no need to explicitly force the future.

We will see below that explicit futures can be implemented in Concurrent Haskell while implicit futures need some primitives which are outside the Concurrent Haskell language.

The advantage of futures is their easy use: for a lot of applications futures can be used as basic concurrency primitive without explicitly taking care about the synchronization of concurrent threads. Moreover, the futures perform this synchronization automatically. Futures can also be used in functional-logic programming to model the unknown value of logical variables as e.g in Mozart [Moz11].

Concurrent Haskell was proposed in [PGF96], but its current implementation in the Glasgow Haskell Compiler is slightly modified. We refer to the current implementation in the GHC (a description can also be found in [Pey01,PS09]), and give a short overview of some basic constructs of Concurrent Haskell.

Concurrent Haskell extends Haskell by a primitive `forkIO` and by synchronizing variables `MVar`. `MVar`s behave like single one-place buffers: `MVar`s are either empty or filled. The primitive operation `newEmptyMVar` creates an empty `MVar`. The operation `takeMVar` reads the value of a filled `MVar` and empties it. All threads that want to execute `takeMVar` on this empty `MVar` are blocked until the `MVar` becomes filled again. Similarly, `putMVar` v e writes the expression e into the `MVar` v, if v is empty, and blocks otherwise until the `MVar` becomes empty. The primitive for thread creation in Concurrent Haskell is `forkIO :: IO () -> IO ThreadId`. Applied to an IO-action, a concurrent thread is immediately started to compute the action concurrently. From the perspective of the calling thread, the result is a unique identifier of the concurrent thread, which for instance can be used to kill the concurrent thread using `killThread`. As already observable by the type of `forkIO`, the result of the concurrently started IO-action must be the unit type `()` (packed into the `IO`-monad), and the result of `forkIO` itself is only a thread identifier.

Explicit Futures can be implemented in Concurrent Haskell using `forkIO` and `MVar`s. An implementation in Haskell is:

```
type EFuture a = MVar a

efuture :: IO a → IO (EFuture a)
efuture act =
 do  ack ← newEmptyMVar
```

```
    forkIO (act >>= putMVar ack)
    return ack

force :: EFuture a → IO a
force x = takeMVar x >>= (λr → putMVar x r >> return r)
```

An explicit future is represented by an `MVar`. The creation of an explicit future first creates an empty `MVar` and then starts the computation of the action corresponding to the future in a concurrent thread such that after finishing the computation the result is written into the empty `MVar`. From the view of the calling thread a future in form of an `MVar` is immediately returned. If the value of the future is needed then the future must be forced explicitly by calling `force` which reads the `MVar`. If the future value is not computed already, then a wait situation arises until the concurrent computation is finished.

Note, that programming with explicit futures is often uncomfortable, since the programmer must be careful to explicitly force the future at the right time. It is more desirable that the future gets (automatically) forced when it is needed through data dependencies such that the programmer does not need to care about explicit forces. Unfortunately, this behavior is not implementable using explicit futures.

Implicit Futures can be implemented by using a well-known technique to delay the computation of a sequential monadic IO-computation: We use Haskell's `unsafeInterleaveIO` which is well-used to delay computations in the IO-monad and to break sequentiality (i.e. to implement *lazy IO*) (see e.g. [PW93,Pey01]). For instance, the standard implementation of `readFile` for lazy file reading uses `unsafeInterleaveIO` to delay the reading of the single characters of a file. An implementation of implicit futures is as follows:

```
future :: IO a → IO a
future code = do ack ←newEmptyMVar
                 thread ← forkIO (code >>= putMVar ack)
                 unsafeInterleaveIO (do result ← takeMVar ack
                                        killThread thread
                                        return result)
```

First an empty `MVar` is created, which will be used to store the result of the concurrent computation. This computation is created using `forkIO` which writes its result into the future when it becomes available. The last part consists of taking the result, killing the concurrent thread and returning the result. This part is delayed using `unsafeInterleaveIO`. Note, that without the use of `unsafeInterleaveIO` the calling thread would be blocked until the concurrent computation has finished which would not implement the desired behavior of futures. For the shown implementation the calling thread only becomes blocked if it demands the result of an unevaluated future.

Thus it is possible to implement implicit futures in Haskell using the `unsafeInterleaveIO`-primitive (which is outside the Haskell-Standard). But the general use of `unsafeInterleaveIO` breaks referential transparency, since impure effects may become visible using pure functions. Nevertheless we believe

that the use in the encoding of futures is "safe". In this paper we make a first step in showing this claim by analyzing the calculus CHF (Concurrent Haskell with Futures): We will show that the usual laws like indifference of call-by-need and call-by-name evaluation and the correctness of the monad laws are valid for CHF.

The Calculus CHF We investigate the extension of Concurrent Haskell where the above `future`-operation is built-in as a primitive using the calculus CHF as a model. CHF is a process calculus which comprises (unlike the π-calculus [Mil99,SW01]) shared memory in form of Haskell's MVars, threads (i.e. futures) and heap bindings. On the expression level we allow monadic IO-computations as well as usual pure functional expressions extending the lambda calculus by data constructors, case-expressions, recursive let-expressions, as well as Haskell's `seq`-operator for sequential evaluation. We add a monomorphic type system to CHF with recursive types and where polymorphic data constructors are monomorphically instantiated. Since we want to keep the formalism and proofs simple, we keep the type system as small as possible, nevertheless we believe that our results are transferable to a polymorphic type system. We present an operational semantics for CHF as a (call-by-need) small-step reduction relation (called *standard reduction*) where the monadic operations are performed as rewriting steps which relieves us from the issue how to implement the bind-operator in Haskell (those correctness issues are analyzed e.g. in [AS98]). That is we follow a suggestion made by Simon Peyton Jones in [Pey01] and add the bind-operator as a *primitive* of the language. We will show in this paper that CHF has a well-behaved semantical underpinning. Our calculus is closely related to the process calculus presented in [Pey01] where the differences are: We provide an operational semantics for the monadic and the functional part while [Pey01] assumes an a priori given denotational semantics for functional expressions. We do not model the `delay`-operator and external input and output, and thus use an unlabeled reduction while [Pey01] uses a labelled transition system.

Compared to threads in Concurrent Haskell, CHF does not include a primitive to kill running threads, which is reasonable since threads are futures which may be referenced somewhere else. In CHF a *successfully evaluated* thread will become a usual heap binding, that is the result is kept while the thread is removed. Running threads that do not longer contribute to the final result can be garbage collected.

As program equivalence we will use *contextual equivalence* (see e.g. [Mor68,Plo75]), that is two programs are equal iff their observable behavior is indistinguishable even if the programs are used as a subprogram of any other program (i.e. if the programs are plugged into any arbitrary context). For nondeterministic and concurrent programming languages it is usually not enough to observe termination, only. Thus we will use a combination of two tests: Can a program terminate (called *may-convergence*) and does a program never loose the ability to converge (called *should-convergence*, or sometimes must-convergence, see e.g. [CHS05,NSSSS07,RV07,SSS08])? In the literature there is often another test used (instead of should-convergence), called

must-convergence (for instance, [DH84]), which holds if a program terminates along all possible computation paths. The difference between should- and must-convergence is that should-convergence is insensible w.r.t. *weakly divergent* programs [NC95], i.e. programs that may run infinitely long but always may terminate along another computation path are should-convergent (but not must-convergent).

Nevertheless this difference is small and we believe that correctness of commonly used program transformations is valid for both predicates. Some advantages of should-convergence (compared to must-convergence) are that restricting the evaluator to *fair scheduling* does not modify the predicate and also not the contextual equivalence, that the equivalence based on may- and should-convergence is invariant under a whole class of test-predicates (see [SSS10]), and inductive reasoning is available as a tool to prove should-convergence.

Results We provide a semantic foundation for Concurrent Haskell extended by futures. In detail we prove a context lemma for expressions which is a helpful tool to prove that expressions are contextually equal. We show that all reduction rules are *correct program transformations* (i.e. they do not change the contextual semantics) except for the rules which take or put an expression from or into an MVar (which are in general incorrect). Using the technique of rewriting on infinite trees (see e.g. [KKSdV97,SS07]), we show that the (call-by-need) standard reduction can be replaced by a call-by-name reduction, which also implies that inlining of expressions is a correct program transformation. Optimizations that are based on sharing or unsharing followed by partial evaluation without take/put on MVars are thus justified by the semantics. We show that (infinite) fairness of reductions can be enforced without changing the contextual semantics based on may-and should-convergence. Finally, we show that our implementation of the IO-monad in CHF satisfies the monad laws if the `seq`-operator's first argument is restricted to non IO-types. This justifies the correctness of using the do-notation and its usual compilation.

Related Work Concurrent futures in Multilisp and their applications are discussed e.g. in [BH77,Hal85]. Our calculus CHF is also related to the (call-by-value) lambda-calculus with futures $\lambda(\mathsf{fut})$ ([NSS06]) which models the core language of Alice ML [Ali11] and has concurrent futures similar to ours, in [NSSSS07] a program equivalence based on contextual equivalence with may- and should-convergence is defined for $\lambda(\mathsf{fut})$ and a set of program transformations is shown correct. In [SSSSN09] variants of $\lambda(\mathsf{fut})$ are presented and their equivalence is shown. In difference to CHF, the calculus $\lambda(\mathsf{fut})$ is a model of an impure programming language, and thus there is no distinction between functional and imperative computations. Moreover, $\lambda(\mathsf{fut})$ has so-called *lazy futures*, which are not included in CHF.

[FF99] present a semantics for a (pure) call-by-value calculus extended with futures and analyze an optimization in the abstract machine, to avoid unnecessary dereferencing operations on evaluated futures (so called "touches"). Since their calculus has no side-effects and futures compute functional expressions, the

futures are different from our futures, but they are similar to Haskell's `par` operator. In [BFKT00] an operational (and a denotational) semantics for Glasgow parallel Haskell [Gla11] is presented. They analyze a pure functional non-strict language extended with an `par`-operator. `par` can be seen as an annotation, that implements explicit parallelism i.e. in (`par` e_1 e_2) the expression e_1 can be evaluated in parallel, while e_2 is the result of the `par`-expression. Thus `par` implements futures for pure functional expressions, e.g. consider the expression `let` $x = e_1$ `in par` x e_2. Since x (and thus e_1) can be evaluated in parallel, one can view x as a future for the value of e_1. Programming with parallel Haskell using strategies was proposed in [THLP98] and recently redesigned in [MML+10].

Finally, [PS09] gives an overview to several techniques for parallel and concurrent programming in Haskell, i.e. into Parallel Haskell, Concurrent Haskell, and Software Transactional Memory [HMPJH05].

A parallel extension of Haskell using processes, but no explicit concurrency, is the programming language Eden [LOMPM05].

Outline In Section 2 we introduce the syntax of the calculus CHF. In Section 3 the operational semantics in form of a small-step reduction relation implementing the call-by-need strategy is defined for CHF. In Section 4 we define contextual equivalence for CHF, we show that this program equivalence remains unchanged if fair evaluation is used and we prove a context lemma for expressions. In Section 5 some first correctness results on program transformations are shown. In Section 6 we show that call-by-name evaluation is correct for CHF and prove correctness of a general copy rule. In Section 7 we use the developed techniques and results to show correctness of the monad laws in CHF. Finally, we conclude in Section 8

2 Syntax and Typing of CHF

In this section we present the syntax of the calculus CHF and provide a type system for the underlying language. The syntax has two layers: On the top-layer are processes and the second layer are expressions. Processes may have expressions as subterms. Let *Var* be a countably infinite set of variables. We denote variables with u, w, x, y, z (maybe indexed by natural numbers).

2.1 Syntax of Processes

The syntax of *processes Proc* is given by the following grammar where $e \in Expr$ is an arbitrary expression (defined below):

$$
\begin{array}{rcll}
P, Q, P_i, Q_i \in \mathit{Proc} & ::= & P_1 \mid P_2 & \text{(parallel composition)} \\
& \mid & x \Leftarrow e & \text{(concurrent thread)} \\
& \mid & \nu x.P & \text{(name restriction)} \\
& \mid & x\,\mathbf{m}\,e & \text{(filled MVar)} \\
& \mid & x\,\mathbf{m}\,- & \text{(empty MVar)} \\
& \mid & x = e & \text{(binding)}
\end{array}
$$

We give an informal meaning of these language constructs: Parallel composition and name restriction act like the corresponding constructs in the π-calculus, i.e. parallel composition constructs concurrently running threads (or other components) and 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. We call the variable x the *thread identifier* or alternatively the *future* x. There is no guarantee that all such threads will eventually be evaluated, an aspect which will be discussed later. MVars are mutable variables which behave like one place buffers, i.e. if a thread wants to fill an already filled MVar, the thread blocks, and a thread also blocks if it tries to take something from an empty MVar. In $x\,\mathbf{m}\,e$ or $x\,\mathbf{m}-$ we call x the *name of the MVar*. Bindings $x = e$ model the global heap of shared expressions, where we say x is a *binding variable*. For a process P we say a variable x is an *introduced variable* if x is a thread identifier, 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.

2.2 Syntax of Expressions

We assume that the syntax contains a set of *data constructors* c which is partitioned into sets, such that each family represents a type T. For a fixed type T we assume that the corresponding data constructors are ordered (denoted with $c_1, \ldots, c_{|T|}$, where $|T|$ is the number of constructors belonging to type T). Each data constructor c has a fixed arity $\mathrm{ar}(c) \geq 0$. For examples we assume that we have a type `Bool` with data constructors `True`, `False` and a type `List` with constructors `Nil` and `:` (written infix as in Haskell).

The syntax of expressions is shown in Fig. 1. Expressions *Expr* comprise the constructs of a usual call-by-need lambda calculus and *monadic expressions* $\mathit{MExpr} \subseteq \mathit{Expr}$ which are used to model IO-operations (inside the IO-monad) by built-in primitives. We explain the syntactic constructs and fix some side conditions: The functional language has the usual constructs of the lambda calculus, i.e. variables, *abstractions* $\lambda x.e$, and *applications* $(e_1\ e_2)$. It is extended by *constructor applications* $(c\ e_1\ \ldots\ e_{\mathrm{ar}(c)})$ which allow constructors to occur fully saturated, only. As selectors `case`-*expressions* are part of the language, where for every type T there is one $\texttt{case}_T$-construct. We sometimes abbreviate `case`-expressions with $\texttt{case}_T\ e\ \texttt{of}\ \mathit{Alts}$ 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. The left hand side $c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})}$ of a `case`-alternative is called a *pattern* where the variables $x_1, \ldots, x_{\mathrm{ar}(c_{T,i})}$ must be pairwise distinct. In the alternative $(c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i)$ the variables x_i become bound with scope e_i. In examples we will also use `if-then-else`-expressions written as `if` e `then` e_1 `else` e_2. These expressions are an abbreviation for the `case`-expression `case` e `of` (`True` $\to e_1$) (`False` $\to e_2$). A further construct of the language are `seq`-*expressions* (`seq` $e_1\ e_2$) which model Haskell's `seq`-operator for strict evaluation. Finally the language has `letrec`-*expressions* which implement local sharing and enables one to declare recursive bindings.

$$
\begin{aligned}
e, e_i \in \mathit{Expr} ::= {} & x \mid \mathit{me} \mid \lambda x.e \mid (e_1\ e_2) \mid c\ e_1 \ldots e_{\mathrm{ar}(c)} \mid \texttt{seq}\ e_1\ e_2 \\
& \mid \texttt{case}_T\ e\ \texttt{of}\ (c_{T,1}\ x_1 \ldots x_{\mathrm{ar}(c_{T,1})} \to e_1) \ldots \\
& \qquad\qquad (c_{T,|T|}\ x_1 \ldots x_{\mathrm{ar}(c_{T,|T|})} \to e_{|T|}) \\
& \mid \texttt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e \quad \text{where } n \geq 1 \\
\mathit{me} \in \mathit{MExpr} ::= {} & \texttt{return}\ e \mid e_1\ \texttt{>>=}\ e_2 \mid \texttt{forkIO}\ e \\
& \mid \texttt{takeMVar}\ e \mid \texttt{newMVar}\ e \mid \texttt{putMVar}\ e_1\ e_2
\end{aligned}
$$

Fig. 1. Syntax of Expressions

In $\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e$ the variables $x_1, \ldots, x_n$ must be 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 sometime abbreviate `letrec`-environments as *Env*, i.e. we write $\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ e$. We finally explain the monadic primitives: The constructs `newMVar`, `takeMVar`, and `putMVar` are used to create and access MVars. The primitive "bind" operator `>>=` implements the sequential composition of IO-operations, the `forkIO`-operator is used for thread creation, and the `return`-operator lifts expressions to monadic expressions. Note that all these primitives must occur with all their arguments present.

Functional values are defined as abstractions and constructor applications. The monadic expressions $(\texttt{return}\ e)$, $(e_1\ \texttt{>>=}\ e_2)$, $(\texttt{forkIO}\ e)$, $(\texttt{takeMVar}\ e)$, $(\texttt{newMVar}\ e)$, $(\texttt{putMVar}\ e_1\ e_2)$ where e, e_i are arbitrary expressions are called *monadic values*. A *value* is either a functional value or a monadic value.

2.3 Well-Formedness, the Distinct Variable Convention and Structural Congruence

We assume that for a process at most one thread is labeled with "main" (i.e. as notation we use $x \overset{\mathsf{main}}{\Longleftarrow} e$). We call this thread the *main thread*. A process is *well-formed*, if all introduced variables are pairwise distinct, and there exists at most one main thread $x \overset{\mathsf{main}}{\Longleftarrow} e$.

On the expression layer variable binders are introduced by abstractions, `letrec`-expressions, and `case`-alternatives, and on the process layer by name restriction $\nu x.P$. This induces a notion of free and bound variables as well as α-renaming and α-equivalence (denoted by $=_\alpha$) on the process and on the expression layer. With $FV(P)$ ($FV(e)$, resp) we denote the free variables of process P (expression e, resp.). We assume the *distinct variable convention* to hold, i.e. free variables are distinct from bound variables, and bound variables are pairwise distinct. We also assume that reductions implicitly perform α-renaming to obey this convention.

For processes we define a structural congruence to equate obviously equal processes, i.e. structural congruence allows one to interchange parallel processes, interchange and move ν-binders, and to α-rename processes:

Definition 2.1. Structural congruence $\equiv$ *is the least congruence satisfying the equations:*

$$\begin{aligned} P_1 \mid P_2 &\equiv P_2 \mid P_1 \\ (P_1 \mid P_2) \mid P_3 &\equiv P_1 \mid (P_2 \mid P_3) \\ (\nu x.P_1) \mid P_2 &\equiv \nu x.(P_1 \mid P_2) \quad \textit{if } x \notin FV(P_2) \\ \nu x_1.\nu x_2.P &\equiv \nu x_2.\nu x_1.P \\ P_1 &\equiv P_2 \qquad\qquad \textit{if } P_1 =_\alpha P_2 \end{aligned}$$

2.4 A Monomorphic Type System

In this section we provide a type system for CHF which mainly distinguishes between processes, functional expressions and monadic expressions. For simplicity we choose a monomorphic type system, (types must be invariant during reduction) for correctness proofs in later sections. If we would use a polymorphic type system then this would require more effort (e.g. one could use a system F like type-system, but there are also other approaches using explicit type labels, for instance [SSSH09]).

Nevertheless we "overload" the data constructors and thus we assume that data types used in case-constructs have a fixed arity, and that the data constructors of every type have a polymorphic type according to the usual conventions. In the language the constructors are used monomorphic. The set of monomorphic types of constructor c is denoted as $\texttt{types}(c)$.
The syntax of types is:

$$\tau ::= \texttt{IO}\ \tau \mid (T\ \tau_1\ \ldots\ \tau_n) \mid \texttt{MVar}\ \tau \mid \tau_1 \to \tau_2$$

Here $(\texttt{IO}\ \tau)$ means that an expression of type τ is packed into a monadic action, and $(\texttt{MVar}\ \tau)$ stands for an `MVar`-reference with content type τ. $\tau_1 \to \tau_2$ is a function type.

To fix the types during reduction, we assume that every variable is explicitly typed, i.e. we assume that every variable x has a built-in type. We denote the global typing function for variables with Γ, i.e. $\Gamma(x)$ 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. For processes the notation $\Gamma \vdash P :: \texttt{wt}$ means that the process P can be well-typed using the global typing function Γ.

The typing rules are in Figure 2. Note that we disallow a reference type or an `IO`-type for the first argument of `seq`-expressions. This restriction is not valid in Haskell, but is indispensable for the validity of several semantical rules and the correctness of program transformations, like the monad laws (see Section 7). Note that the type system can easily be transformed into a "more standard one" if Γ is viewed as a type environment and the rules for variable binders are adjusted such that they add type assumptions to the environment. Note also that for our type system it is essential that processes fulfill the distinct variable convention before type checking is performed. For instance, the process $z_1 \Leftarrow (\lambda x.\texttt{return}\ x)\ \texttt{True} \mid z_2 \Leftarrow (\lambda x.\texttt{return}\ x)\ \texttt{Nil}$ cannot be typed (since the type of x in both abstractions is different), while $z_1 \Leftarrow (\lambda x.\texttt{return}\ x)\ \texttt{True} \mid z_2 \Leftarrow (\lambda x'.\texttt{return}\ x')\ \texttt{Nil}$ can be typed.

$$\frac{\Gamma \vdash e :: \texttt{IO}\ \tau}{\Gamma \vdash x \Leftarrow e :: \texttt{wt}} \quad \frac{\Gamma \vdash e :: \tau}{\Gamma \vdash x = e :: \texttt{wt}} \quad \frac{\Gamma \vdash P_1 :: \texttt{wt}, \quad \Gamma \vdash P_2 :: \texttt{wt}}{\Gamma \vdash P_1 \mid P_2 :: \texttt{wt}}$$

$$\frac{\Gamma(x) = \texttt{MVar}\ \tau, \quad \Gamma \vdash e :: \tau}{\Gamma \vdash x\, \mathbf{m}\, e :: \texttt{wt}} \quad \frac{\Gamma(x) = \texttt{MVar}\ \tau}{\Gamma \vdash x\, \mathbf{m} - :: \texttt{wt}} \quad \frac{\Gamma \vdash P :: \texttt{wt}}{\Gamma \vdash \nu x.P :: \texttt{wt}}$$

$$\frac{\Gamma \vdash e :: \tau}{\Gamma \vdash \texttt{return}\ e :: \texttt{IO}\ \tau} \quad \frac{\Gamma \vdash e_1 :: \texttt{IO}\ \tau_1, \quad \Gamma \vdash e_2 :: \tau_1 \to \texttt{IO}\ \tau_2}{\Gamma \vdash e_1 \texttt{>>=} e_2 :: \texttt{IO}\ \tau_2} \quad \frac{\Gamma \vdash e :: \texttt{IO}\ \tau}{\Gamma \vdash \texttt{forkIO}\ e :: \texttt{IO}\ \tau}$$

$$\frac{\Gamma \vdash e :: \texttt{MVar}\ \tau}{\Gamma \vdash \texttt{takeMVar}\ e :: \texttt{IO}\ \tau} \quad \frac{\Gamma \vdash e_1 :: \texttt{MVar}\ \tau, \quad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash \texttt{putMVar}\ e_1\ e_2 :: \texttt{IO}\ ()} \quad \frac{\Gamma \vdash e :: \tau}{\Gamma \vdash \texttt{newMVar}\ e :: \texttt{IO}\ (\texttt{MVar}\ \tau)}$$

$$\frac{\forall i : \Gamma \vdash e_i :: \tau_i, \quad \tau_1 \to \ldots \to \tau_n \to \tau_{n+1} \in \texttt{types(c)}}{\Gamma \vdash (c\ e_1\ \ldots\ e_{\mathrm{ar}(c)}) :: \tau_{n+1}} \quad \frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2, \quad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash (e_1\ e_2) :: \tau_2}$$

$$\frac{\Gamma(x) = \tau_1, \quad \Gamma \vdash e :: \tau_2}{\Gamma \vdash (\lambda x.e) :: \tau_1 \to \tau_2} \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau} \quad \frac{\begin{array}{c}\Gamma \vdash e_1 :: \tau_1, \quad \Gamma \vdash e_2 :: \tau_2 \\ \tau_1 = \tau_3 \to \tau_4 \text{ or } \tau_1 = (T\ldots)\end{array}}{\Gamma \vdash (\texttt{seq}\ e_1\ e_2) :: \tau_2}$$

$$\frac{\Gamma \vdash e :: \tau_1 \ \text{ and } \tau_1 = (T\ \ldots), \quad \forall i : \Gamma \vdash (c_i\ x_{1,i}\ \ldots\ x_{n_i,i}) :: \tau_1, \quad \forall i : \Gamma \vdash e_i :: \tau_2}{\Gamma \vdash (\texttt{case}_T\ e\ \texttt{of}(c_1\ x_{1,1}\ \ldots\ x_{n_1,1} \to e_1) \ldots (c_m\ x_{1,m}\ \ldots\ x_{n_m,m} \to e_m)) :: \tau_2}$$

$$\frac{\forall i : \Gamma(x_i) = \tau_i, \quad \forall i : \Gamma \vdash e_i :: \tau_i, \quad \Gamma \vdash e :: \tau}{\Gamma \vdash (\texttt{letrec}\ x_1 = e_1,\ \ldots\ x_n = e_n\ \texttt{in}\ e) :: \tau}$$

Fig. 2. Typing rules

Definition 2.2. *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.*

3 Operational Semantics of CHF

In this section we define the operational semantics of the calculus CHF as a small-step reduction relation called *standard reduction*. As a first definition we introduce *successful processes*, i.e. processes which are seen as successful outcomes of the standard reduction.

Definition 3.1. *A well-formed process P is* successful, *if P has a main thread of the form $x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ e$, i.e. $P \equiv \nu x_1 \ldots \nu x_n.(x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ e \mid P')$.*

We allow standard reductions only for well-formed processes which are not successful, i.e. successful as well as non-well-formed processes are irreducible by definition. This can be justified as follows: Non-well-formed processes can be

singled out by the parser of a compiler, successful processes may have reducible threads (but not in the main thread $x \overset{\mathsf{main}}{\Longleftarrow} e$), but in Haskell all concurrent threads are terminated, if the main-thread terminates.

For the definition of the standard reduction we require the notion of contexts. In general a context is an expression with a hole $[\cdot]$, i.e. a special constant which occurs once in the expression. We assume that the hole $[\cdot]$ is typed and carries a type label, which we write as $[\cdot^\tau]$ if we want to make the type explicit. The typing rules are accordingly extended by the rule for the hole:

$$\frac{}{\Gamma \vdash [\cdot^\tau] :: \tau}$$

Given a context $C[\cdot^\tau]$ and an expression $e :: \tau$, $C[e]$ denotes the result of replacing the hole in C wit expression e, where a variable capture is permitted. Since our syntax has different syntactic categories, we require different contexts:

- Process contexts that are processes with a hole at process position.
- Expression contexts that are expressions with a hole at expression position.
- Process contexts with an expression hole, i.e. processes with a hole at expression position.

On the process level we define the *process contexts PCtxt* as follows, where $P \in Proc$:

$$\mathbb{D}, \mathbb{D}_i \in PCtxt ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}$$

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
- a thread and a set of bindings (which are referenced and used by the selected thread)

Although we require further classes of contexts for the complete definition of the standard reduction, we introduce the standard reduction at this point. We will then explain further contexts and thereafter we explain the reduction rules in detail.

Definition 3.2. *The* standard reduction rules *are given in Fig. 3 where the outer PCtxt-context is omitted. But we assume reductions to be closed w.r.t. PCtxt-contexts and w.r.t. structural congruence, i.e. the* standard reduction relation $\xrightarrow{sr}$ *is the union of the rules in Fig. 3 and if* $P_1 \equiv \mathbb{D}[P_1']$ *and* $P_2 \equiv \mathbb{D}[P_2']$ *such that* $P_1' \xrightarrow{sr} P_2'$*, then also* $P_1 \xrightarrow{sr} P_2$*.*

With $\xrightarrow{sr,+}$ *we denote the transitive closure of* $\xrightarrow{sr}$*, and with* $\xrightarrow{sr,*}$ *we denote the reflexive-transitive closure of* $\xrightarrow{sr}$*.*

Monadic Computations

(lunit) $y \Leftarrow \mathbb{M}[\texttt{return}\ e_1\ \texttt{>>=}\ e_2] \xrightarrow{sr} y \Leftarrow \mathbb{M}[e_2\ e_1]$

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

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

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

(fork) $y \Leftarrow \mathbb{M}[\texttt{forkIO}\ e] \xrightarrow{sr} \nu z.(y \Leftarrow \mathbb{M}[\texttt{return}\ z] \mid z \Leftarrow e)$
where z is fresh and the created thread is not the main thread

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

Functional Evaluation

(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

(cpcx) $\widehat{\mathbb{L}}[x] \mid x = c\ e_1 \ldots\ e_n$
$\xrightarrow{sr} \nu y_1, \ldots y_n.(\widehat{\mathbb{L}}[c\ y_1\ \ldots\ y_n] \mid x = c\ y_1\ \ldots\ y_n \mid y_1 = e_1 \mid \ldots \mid y_n = e_n)$
if c is a constructor, or `return`, `>>=`, `takeMVar`, `putMVar` `newMVar`, or `forkIO`

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

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

(case) $\mathbb{L}[\texttt{case}_T\ (c\ e_1\ \ldots\ e_n)\ \texttt{of}\ \ldots((c\ y_1\ \ldots\ y_n) \to e)\ldots]$
$\xrightarrow{sr} \nu x_1, \ldots, x_n.(\mathbb{L}[e] \mid y_1 = e_1 \mid \ldots \mid y_n = e_n])$ if $n > 0$

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

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

Fig. 3. Standard reduction rules

For the evaluation of monadic expressions we define the *monadic contexts MCtxt*. They are used to "find" the first monadic action in a sequence of actions.

$$\mathbb{M}, \mathbb{M}_i \in \mathit{MCtxt} ::= [\cdot] \mid \mathbb{M}\ \texttt{>>=}\ e$$

On expressions we use usual (call-by-name) *expression evaluation contexts ECtxt* defined as follows:

$$\mathbb{E}, \mathbb{E}_i \in \mathit{ECtxt} ::= [\cdot] \mid (\mathbb{E}\ e) \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ \mathit{alts}) \mid (\texttt{seq}\ \mathbb{E}\ e)$$

Sometimes, the evaluation of the (first) argument of the monadic operations `takeMVar` and `putMVar` must be forced. (i.e. before the corresponding monadic action can be performed). For example, the process

$$x \Leftarrow (\texttt{takeMVar}\ ((\lambda x.x)\ y))\ \texttt{>>=}\ \lambda z.(\texttt{return}\ ())) \mid y\,\mathbf{m}\,\texttt{True}$$

must first evaluate the subexpression $((\lambda x.x)\ y)$ before performing the `takeMVar`-operation. To model these cases correctly (i.e. as in Haskell) we introduce the *forcing contexts FCtxt*.

$$\mathbb{F}, \mathbb{F}_i \in \mathit{FCtxt} ::= \mathbb{E} \mid (\texttt{takeMVar}\ \mathbb{E}) \mid (\texttt{putMVar}\ \mathbb{E}\ e)$$

Finally, we define the contexts LCtxt which model the search for a redex after one thread was already selected. The necessary reduction may either be a monadic computation, or a "functional evaluation". If the thread needs the value of a binding, then the functional evaluation may be performed inside a binding. For instance consider the process

$$x \overset{\mathsf{main}}{\Longleftarrow} y\ \texttt{True} \mid y = z\ \texttt{False} \mid z = (\lambda x_1.\lambda x_2.\lambda x_3.\texttt{return}\ x_3)\ \texttt{False}$$

The main-thread needs the value of y, the binding for y needs the result of z. Hence, the standard reduction is:

$$\xrightarrow{sr,lbeta} x \overset{\mathsf{main}}{\Longleftarrow} y\ \texttt{True} \mid y = z\ \texttt{False} \mid z = \lambda x_2.\lambda x_3.\texttt{return}\ x_3 \mid x_1 = \texttt{False}$$

The contexts LCtxt model this redex search, they are defined as follows:

$$\begin{aligned}\mathbb{L}, \mathbb{L}_i \in \mathit{LCtxt} ::= {} & x \Leftarrow \mathbb{M}[\mathbb{F}] \\ \mid {} & x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1 \\ & \text{where } \mathbb{E}_2, \ldots \mathbb{E}_n \text{ are not the empty context.}\end{aligned}$$

For the copying rules (i.e. the rules (cp) and (cpcx)) we define a special class of LCtxt-contexts, the contexts $\widehat{\mathit{LCtxt}}$, which require that the context $\mathbb{E}_1$ must not be empty

$$\begin{aligned}\widehat{\mathbb{L}}, \widehat{\mathbb{L}}_i \in \widehat{\mathit{LCtxt}} ::= {} & x \Leftarrow \mathbb{M}[\mathbb{F}] \\ \mid {} & x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1 \\ & \text{where } \mathbb{E}_1, \mathbb{E}_2, \ldots \mathbb{E}_n \text{ are not the empty context.}\end{aligned}$$

This distinction is necessary for the case of variable-to-variable bindings, i.e. if a thread demands the value of x and $x = y$ is a binding, then evaluation does not follow this binding, but copies the name y. For instance, for the process

$$z \overset{\mathsf{main}}{\Longleftarrow} x \mid x = y \mid y = \texttt{return}\ ()$$

the standard reduction proceeds as follows:

$$\begin{aligned}&\xrightarrow{sr,cp} z \overset{\mathsf{main}}{\Longleftarrow} y \mid x = y \mid y = \texttt{return}\ () \\ &\xrightarrow{sr,cpcx} z \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ w \mid x = y \mid y = \texttt{return}\ w \mid w = ()\end{aligned}$$

We will now explain the standard reduction rules of Fig. 3 in detail. The rules are divided into two sets of reductions: The first part of the rules performs monadic computations while the second part performs functional evaluation on the expression level. The *redex* is the subexpression together with its position defined as follows: For (lunit), (tmvar), (pmvar),(nmvar), (fork), it is the monadic expression in the context $\mathbb{M}$, for the rule (unIO), it is $y \Leftarrow \texttt{return}\ e$, for (mkbinds), (lbeta), (case), (seq), it is the functional expression in the context $\mathbb{L}$, and for (cp), (cpcx) it is the variable x in the context $\widehat{\mathbb{L}}$.

The rule (lunit) is the direct implementation of the monadic sequencing operator `>>=`: Consider a sequence a `>>=` b. If a is of the form `return` e then the monadic computation of a is finished (with the result e), hence the next computation (b) of the sequence can be started. Since the result e of the first computation may be used by b, the evaluation proceeds with (b e).

The rules (tmvar) and (pmvar) perform a `takeMVar`- or `putMVar`-operation on a filled (or empty, resp.) MVar. Note that there is no rule for a `takeMVar`-operation on an empty MVar (and also no rule for a `putMVar`-operation on a filled MVar), which models the blocking behavior of MVars. The rule (nmvar) creates a new filled MVar.

The rule (fork) spawns a new thread for a concurrent computation. In Haskell the return value of a `forkIO`-operation is a thread identifier (usually a number). Since our model uses variables to identify threads, the corresponding variable is returned.

The rule (unIO) binds the result of a monadic computation to a functional binding, i.e. the value of a concurrent future becomes accessible.

The rules (cp) and (cpcx) are used to inline a demanded binding $x = e$. Here e must be an abstraction, a variable, a constructor application or a monadic expression. For the correct treatment of call-by-need evaluation for constructor applications (c e_1 $\ldots$ e_n) (and also for monadic expressions) the (maybe non-value) arguments are shared by new bindings.

The rule (mkbinds) moves the bindings of a `letrec`-expression into the global bindings. ν-binders are introduced to restrict the access to the bindings of the concurrent thread only. The rule (lbeta) is the call-by-need variant of classical β-reduction, where the argument is not substituted in the body of the abstraction but shared by a new global binding. The (case)-reduction reduces a `case`-expression, where – if the scrutinee is not a constant – also bindings are created to implement sharing. The (seq)-rules evaluate a `seq`-expression: If the first argument is a functional value, then the seq-expression is replaced by its second argument.

Proposition 3.3. *The following properties hold for the standard reduction* $\xrightarrow{sr}$*:*

- *If* $P \xrightarrow{sr} P'$ *and* P *is well-formed, then* P' *remains well-formed.*
- *If* $P \xrightarrow{sr} P'$ *and* P *is well-typed, then* P' *remains well-typed.*
- *Reduction is unique for threads. I.e. If* P *contains only one thread, then for all* P_1, P_2 *with* $P \xrightarrow{sr} P_i$ *(i=1,2):* $P_1 \equiv P_2$*.*
- *Reduction cannot introduce or remove a main-thread.*

Proof. The first part holds, since the reduction rules only introduce process identifiers which are fresh and never introduce a main thread. Type preservation holds since every redex keeps the type of subexpressions. The remaining parts can be shown by induction on the process structure.

Example 3.4. The following example shows that standard reduction is non-deterministic. Consider the process P:

$$x \overset{\mathsf{main}}{\Longleftarrow} \texttt{takeMVar}\ y \mid z \Leftarrow \texttt{takeMVar}\ y \texttt{ >>= } \lambda w.(\texttt{putMVar}\ y\ \texttt{False}) \mid y\, \mathbf{m}\, \texttt{True}$$

If first the main-thread is reduced, then we obtain a successful process (we omit ν-binders):

$$P \xrightarrow{sr,tmvar} x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return True} \mid z \Leftarrow \texttt{takeMVar}\ y \texttt{ >>= } \lambda w.(\texttt{putMVar}\ y\ \texttt{False}) \mid y\,\mathbf{m}\,-.$$

If first the thread with identifier z is reduced four times and then the main-thread is reduced, then we also obtain a successful process, but the result is different:

$$\begin{aligned}
P &\xrightarrow{sr,tmvar} x \overset{\mathsf{main}}{\Longleftarrow} \texttt{takeMVar}\ y \mid z \Leftarrow \texttt{return True} \texttt{ >>= } \lambda w.(\texttt{putMVar}\ y\ \texttt{False}) \mid y\,\mathbf{m}\,- \\
&\xrightarrow{sr,lunit} x \overset{\mathsf{main}}{\Longleftarrow} \texttt{takeMVar}\ y \mid z \Leftarrow (\lambda w.(\texttt{putMVar}\ y\ \texttt{False}))\ \texttt{True} \mid y\,\mathbf{m}\,- \\
&\xrightarrow{sr,lbeta} x \overset{\mathsf{main}}{\Longleftarrow} \texttt{takeMVar}\ y \mid z \Leftarrow \texttt{putMVar}\ y\ \texttt{False} \mid w = \texttt{True} \mid y\,\mathbf{m}\,- \\
&\xrightarrow{sr,pmvar} x \overset{\mathsf{main}}{\Longleftarrow} \texttt{takeMVar}\ y \mid z \Leftarrow \texttt{return}\ () \mid w = \texttt{True} \mid y\,\mathbf{m}\,\texttt{False} \\
&\xrightarrow{sr,tmvar} x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return False} \mid z \Leftarrow \texttt{return}\ () \mid w = \texttt{True} \mid y\,\mathbf{m}\,-
\end{aligned}$$

Note that after the first (sr,tmvar)-reduction the main-thread is blocked until the MVar y becomes filled.

Example 3.5. As a further example we demonstrate how a (monadic) binary amb-operator can be implemented:

$$\begin{aligned}
amb = \lambda x_1, x_2.\ & \texttt{newMVar}\ x_1 \texttt{ >>=} \\
& \lambda m.\texttt{takeMVar} \texttt{ >>=} \\
& \lambda_.(\texttt{forkIO}\ (\texttt{seq}\ x_1\ (\texttt{putMVar}\ m\ x_1)) \texttt{ >>=} \\
& \lambda_.(\texttt{forkIO}\ (\texttt{seq}\ x_2\ (\texttt{putMVar}\ m\ x_2)) \texttt{ >>=} \\
& \lambda_.\texttt{takeMVar}\ m
\end{aligned}$$

This expression implements McCarthy's bottom-avoiding choice [McC63], that is applied to two arguments e_1, e_2, the result of $amb\ e_1\ e_2$ is the monadic action returning the value of e_1 or e_2 (if both evaluate to a value), or the value of e_i if e_j diverges (for $(i,j) \in \{(1,2),(2,1)\}$). With futures we can easily extend the binary operator for a whole list of arguments as follows

$$\begin{aligned}
&\texttt{letrec}\ ambList = \lambda xs.\texttt{case}_{List}\ xs\ \texttt{of} \\
&\qquad\qquad (\texttt{Nil} \rightarrow \texttt{return}\ \bot), \\
&\qquad\qquad (y : ys \rightarrow \texttt{forkIO}\ (ambList\ ys) \texttt{ >>=} \lambda ys'.amb\ y\ ys') \\
&\texttt{in}\ ambList
\end{aligned}$$

where $\bot$ is any closed diverging expression, e.g. ($\texttt{letrec}\ x = x\ \texttt{in}\ x$)

4 Program Equivalence and Context Lemmas

In this section we introduce a notion of program equivalence for processes as well as for expressions. We will use contextual equivalence by observing may- and should-convergence. Subsequently, we will prove some context lemmas which ease proofs of equivalences.

4.1 Contextual Equivalence

Contextual equivalence equates two processes P_1, P_2 if their observable behavior is indistinguishable if P_1 and P_2 are plugged into any process context.

For nondeterministic (and also concurrent) calculi the observation of may-convergence, i.e. the question whether or a not a process can be reduced to a successful process, is *not* sufficient to distinguish obviously different processes. It is also necessary to analyze the possibility of introducing errors or non-termination. Thus we will observe may-convergence and a variant of must-convergence which we call should-convergence (see [RV07,SSS08]). The definitions are as follows:

Definition 4.1. *A process P* may-converges *(written as $P{\downarrow}$), iff P is well-formed and P reduces to a successful process, i.e.*

$$P{\downarrow} \text{ iff } P \text{ is well-formed and } \exists P' : P \xrightarrow{sr,*} P' \wedge P' \text{ successful}$$

If a process P is not may-convergent, then P must-diverges *written as $P{\Uparrow}$.*

A process P should-converges *(written as $P{\Downarrow}$), iff P is well-formed and P remains may-convergent under reduction, i.e.*

$$P{\Downarrow} \text{ iff } P \text{ is well-formed and } \forall P' : P \xrightarrow{sr,*} P' \implies P'{\downarrow}$$

If P is not should-convergent then we say P may-diverges *written as $P{\uparrow}$.*

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

In the literature there is one other main notion of must-convergence which requires that all reduction sequences of a process are finite and end successfully. Note that should-convergence allows infinite reduction sequences if the ability to converge is never lost. Although the two notions of must-convergence induce slightly different notions of contextual equivalence, but there appears to be no difference w.r.t. usual program transformations.

Note also that may-divergence can alternatively be characterized by: A process P is may-divergent if there is a finite reduction sequence $P \xrightarrow{sr,*} P'$ such that P' cannot converge, i.e. $P'{\Uparrow}$.

Our definition of reduction implies that non-wellformed processes are always must-divergent, since they are irreducible and never successful. Also, the process construction by $\mathbb{D}[P]$ is always well-typed if P is well-typed, since we assume that variables have a built-in type.

Definition 4.2. *Contextual approximation $\leq_c$ and contextual equivalence $\sim_c$ on processes are defined as follows:*

$$\begin{aligned} P_1 \leq_{\downarrow} P_2 \;\; &\text{iff } \forall \mathbb{D} \in PCtxt : \mathbb{D}[P_1]{\downarrow} \implies \mathbb{D}[P_2]{\downarrow} \\ P_1 \leq_{\Downarrow} P_2 \;\; &\text{iff } \forall \mathbb{D} \in PCtxt : \mathbb{D}[P_1]{\Downarrow} \implies \mathbb{D}[P_2]{\Downarrow} \\ \leq_c &:= \leq_{\downarrow} \cap \leq_{\Downarrow} \\ \sim_c &:= \leq_c \cap \geq_c \end{aligned}$$

Remark 4.3. Let P_1, P_2 be wellformed processes and I_i be the free introduced variables of P_i, for $i = 1, 2$. If P_1 and P_2 do not have a main thread and $I_1 \neq I_2$ then $P_1 \not\sim_c P_2$: W.l.o.g. assume $x \in I_1$ but $x \notin I_2$ and consider the context $\mathbb{D} := y \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}() \mid x = x \mid [\cdot]$ where $y \notin I_1 \cap I_2$. Then $\mathbb{D}[P_2]$ is successful, and thus $\mathbb{D}[P_2]\downarrow$. On the other hand $\mathbb{D}[P_1]\Uparrow$, since $\mathbb{D}[P_1]$ is not well-formed.

The previous definition only equates (or distinguishes) *processes*. We now define contextual approximation and equivalence on *expressions*. Let *CCtxt* be the class of process contexts that have their (typed) hole at an arbitrary expression position. We use $\mathbb{C}, \mathbb{C}_i$ for *CCtxt*-contexts.

Definition 4.4. *Let τ be a type. Contextual approximation $\leq_{c,\tau}$ and contextual equivalence $\sim_{c,\tau}$ on expressions are defined as follows, where e_1, e_2 are expressions of type τ*

$$\begin{array}{rl} e_1 \leq_{\downarrow,\tau} e_2 & \textit{iff } \forall \mathbb{C}[\cdot^\tau] \in \mathit{CCtxt} : \mathbb{C}[e_1]\downarrow \implies \mathbb{C}[e_2]\downarrow \\ e_1 \leq_{\Downarrow,\tau} e_2 & \textit{iff } \forall \mathbb{C}[\cdot^\tau] \in \mathit{CCtxt} : \mathbb{C}[e_1]\Downarrow \implies \mathbb{C}[e_2]\Downarrow \\ \leq_{c,\tau} & := \leq_{\downarrow,\tau} \cap \leq_{\Downarrow,\tau} \\ \sim_{c,\tau} & := \leq_{c,\tau} \cap \geq_{c,\tau} \end{array}$$

Remark 4.5. An interesting fact on the contextual preorder is that closed must-divergent expressions are *not* least elements w.r.t. $\leq_c$. The reason is, that *amb* is definable in CHF (see Example 3.5). Consider the context

$$C := z \overset{\mathsf{main}}{\Longleftarrow} \texttt{amb True } [\cdot] \texttt{ >>= } \lambda r.(\texttt{if } r \texttt{ then return True else } \bot)$$

and let $\bot$ be a closed must-divergent expression of type `Bool`. Then $C[\bot]\Downarrow$, but $C[\texttt{False}]\uparrow$ and thus $\bot \not\leq_{c,\texttt{Bool}} \texttt{False}$.

As a first result we show that structural congruence preserves contextual equivalence:

Proposition 4.6. *Let P_1, P_2 be well-formed processes such that $P_1 \equiv P_2$. Then $P_1 \sim_c P_2$.*

Proof. The claim follows easily from the following two observations:

- For every process context $\mathbb{D}$ and process P_3: $\mathbb{D}[P_1] \xrightarrow{sr} P_3$ iff $\mathbb{D}[P_2] \xrightarrow{sr} P_3$, since $\equiv$ is a congruence and since $\xrightarrow{sr}$ is closed w.r.t. structural congruence.
- $\mathbb{D}[P_1]$ is successful iff $\mathbb{D}[P_2]$ is successful, since structural congruence does not remove nor introduce a main thread.

4.2 Fairness

In this section we show that contextual equivalence is unchanged if we disallow unfair reduction sequences. I.e. we assume that fair scheduling is performed for a real implementation of CHF, but since we will show that contextual equivalence is unchanged we do not need to take care about it in our further reasoning.

We first introduce a notion for all maximal reduction sequences for a given process:

Definition 4.7. *For a process P let $\mathcal{M}(P)$ be the set of all maximal sr-reduction sequences starting with P, i.e. all finite reductions sequences ending in an irreducible process and all infinite reduction sequences. With $\mathcal{M}^\omega(P)$ we denote the reduction sequences of $\mathcal{M}(P)$ which are infinite, and let $\mathcal{M}^*(P) := \mathcal{M}(P) \setminus \mathcal{M}^\omega(P)$.*

We repeat the definitions of may- and should-convergence in terms of $\mathcal{M}$:

- A process P is may-convergent, iff $\mathcal{M}^*(P)$ contains a reduction sequence ending in a successful process.
- A process P is should-convergent, iff all reduction sequences of $\mathcal{M}^*(P)$ end in a successful process and for every infinite reduction sequence $RED_\omega \in \mathcal{M}^\omega(P)$ the following holds: for every finite prefix RED of RED_ω there exists a finite reduction sequence RED' in $\mathcal{M}^*(P)$ such that RED is a prefix of RED'.

For a process $P \equiv \mathbb{D}[x \Leftarrow e]$ we say thread x *is enabled* if there is a standard reduction applicable to P, such that $x \Leftarrow e$ is a part of the redex or e (with its position in thread x) is a superexpression of the redex. In a reduction sequence we say thread x is reduced, if there exists a reduction step where x is enabled and the corresponding standard reduction is used.

Now we define a notion of fairness for reduction sequences:

Definition 4.8. *For a process P a reduction sequence $RED \in \mathcal{M}(P)$ is called* unfair *if there is an infinite suffix RED' of RED and there exists a thread x which is enabled in infinitely many processes of RED' but x is never reduced. Otherwise, we say RED is a* fair *reduction sequence. With $\mathcal{M}_f(P) \subseteq \mathcal{M}(P)$ we denote the set of fair reduction sequences of process P. We use $\mathcal{M}_f^*(P)$ ($\mathcal{M}_f^\omega(P)$, resp.) for the finite (infinite, resp.) sequences of $\mathcal{M}_f(P)$.*

Fair may-convergence *and* fair should-convergence *are defined as may- and should-convergence, where the set $\mathcal{M}(P)$ is replaced by the set $\mathcal{M}_f(P)$.*

The definitions imply that may-convergence and fair may-convergence coincide. We now consider should-convergence. One direction is easy:

Lemma 4.9. *If a process P is should-convergent, then P is fair should-convergent.*

Proof. For finite reductions sequences this obvious, since $\mathcal{M}^*(P) = \mathcal{M}_f^*(P)$. For infinite reductions sequences we have $\mathcal{M}_f^\omega(P) \subseteq \mathcal{M}^\omega(P)$ and thus every prefix RED_1 of a reduction sequence $RED \in \mathcal{M}_f^\omega(P)$ is also a prefix of $RED \in \mathcal{M}^\omega(P)$ and for any $RED' \in \mathcal{M}^*(P)$, that has RED_1 as a prefix, it holds $RED' \in \mathcal{M}_f^*(P)$.

For the other direction we first prove a helpful lemma:

Lemma 4.10. *For every process P there exists a reduction sequence $RED_f \in \mathcal{M}_f(P)$.*

Proof. If there exists a finite reduction sequence RED such that RED is a maximal reduction sequence, then we choose this sequence and we are finished. Otherwise, we use the following scheduling procedure: For every MVar x, we built two FIFO-queues:

- The take-queue for pending `takeMVar`-operations (i.e. a list of the corresponding thread identifiers)
- The put-queue for pending `putMVar`-operations.

We also order all threads in another FIFO-queue $\mathcal{Q}$. Now we define a scheduling on the process together with the queues: Let x be the first thread in $\mathcal{Q}$.

- If the thread x is enabled and the corresponding reduction is not a (sr,tmvar)- or a (sr,pmvar)-reduction, then the thread is reduced and then appended to the end of $\mathcal{Q}$. If the thread x has now (sr,tmvar)- or a (sr,tmvar)-redex for MVar y, then the thread is also appended at the end of the corresponding MVar-queue of y. If a new thread is created by a (sr,fork)-reduction, then this thread is also appended to the end of $\mathcal{Q}$. If the new thread has an (sr,tmvar)- or a (sr,tmvar)-redex for MVar y, then the thread is also appended at the end of the corresponding MVar-queue of y.
- If the thread x is enabled, and a (tmvar)- or (pmvar)-reduction is the corresponding reduction, and the thread x is the first one in corresponding take- or put-queue of the MVar, then the thread is reduced, and the thread identifier is removed from the FIFO-queue of the MVar, otherwise, the thread x is not reduced and appended to the end of $\mathcal{Q}$. If the thread x after the reduction has now (sr,tmvar)- or a (sr,tmvar)-redex for MVar y, then the thread is also appended at the end of the corresponding MVar-queue of y.
- If the thread x is not enabled, or the corresponding reduction is a (tmvar)- or (pmvar)-reduction, but x is not at the front of the corresponding take- or put-queue of the MVar, then the thread is moved to the end of $\mathcal{Q}$, but also remains at its place in the take- or put-queue of the MVar.

Now we argue that this scheduling cannot lead to an unfair reduction sequence starting with P: It is impossible that there is an infinite suffix of the reduction sequence where a thread x is enabled in every process, since the FIFO-queue $\mathcal{Q}$ ensures that x is reduced after finitely many steps. It is also impossible, that a thread x is enabled infinitely often, but disabled finitely often, since then we could choose another infinite suffix where x is enabled in all processes. If a thread x is enabled and disabled infinitely often, then x can only become disabled, if x wants to perform a `takeMVar`- (or a `putMVar`-operation, resp.), and after being enabled another thread has emptied (or filled, resp.) the corresponding MVar, which was in the MVar queue at a place before x. Nevertheless this case is impossible, since the take-queue (put-queue, resp.) on the MVar ensures that after some `putMVar`- and `takeMVar`-operations thread x is the only thread that can access the MVar. These operations must happen, since x must become enabled. Due to the queue $\mathcal{Q}$ thread x is eventually reduced.

Corollary 4.11. *Let RED be a finite sr-reduction sequence starting with process P. Then there exists a reduction sequence $RED_f \in \mathcal{M}_f(P)$ which has RED as prefix.*

Proof. This follows by the previous lemma (Lemma 4.10): We assume that $P \xrightarrow{RED} P'$ and then we extend the reduction sequence by a fair reduction sequence for P'. The derived reduction sequence for P is obviously fair.

Lemma 4.12. *If P is fair should-convergent, then P is should-convergent.*

Proof. The only non-trivial case is: There is a reduction sequence $RED_\omega \in \mathcal{M}^\omega(P)$ with $RED_\omega \notin \mathcal{M}^\omega_f(P)$. Let RED be a (finite) prefix of RED_ω. By Corollary 4.11 there must exist a reduction sequence $RED_1 \in \mathcal{M}_f(P)$ such that RED is a prefix of RED_1. If RED_1 is finite then RED_1 must end in a successful process, and since $\mathcal{M}^*(P) = \mathcal{M}^*_f(P)$ we have $RED_1 \in \mathcal{M}^*(P)$. If RED_1 is infinite then there exists another reduction sequence $RED_2 \in \mathcal{M}^*_f(P)$ which has RED as a prefix and ends in a successful process. Again it also holds that $RED_2 \in \mathcal{M}^*(P)$ and the claim follows.

Theorem 4.13. *Contextual equivalence is unchanged if unfair reduction sequences are forbidden.*

Proof. This follows since the convergence predicates do not change.

Remark 4.14. Note that must-convergence (i.e. the test whether $\mathcal{M}^\omega(P) = \emptyset$) does *not* coincide with *fair* must-convergence (i.e. the test whether $\mathcal{M}^\omega_f(P) = \emptyset$). A counter-example is the (should-convergent) process

$$\begin{array}{rl} P := & x \xLeftarrow{\mathsf{main}} (\lambda x.\ \texttt{return}\ x)\ \texttt{True} \\ \mid & y \Leftarrow \mathit{forever}\ (\texttt{return}\ ()) \\ \mid & \mathit{forever} = \lambda a.a \mathbin{>\!\!>\!=} \lambda\_.(\mathit{forever}\ a) \end{array}$$

$\mathcal{M}^\omega(P) \neq \emptyset$, since an unfair reduction sequence is to always reduce thread y and ignoring the main thread x. On the other hand $\mathcal{M}^\omega_f(P) = \emptyset$, since any fair reduction sequence eventually must reduce the main thread. Note, that successful threads are irreducible by definition.

4.3 Context Lemma for Processes

Definition 4.15. *For a process P we say P is in* prenex normal form *iff $P = \nu x_1, \ldots x_n.(P_1 \mid \ldots \mid P_m)$ and for $1 \leq i \leq m$: P_i does not contain ν-binders.*

Lemma 4.16. *For every process P there exists a process P' with $P \equiv P'$ and P' is in prenex normal form.*

Proof. This follows by structural induction on P.

The following lemma in connection with the previous result implies, that we can more or less ignore ν-binders for reasoning on convergency and thus also on contextual equivalence.

Lemma 4.17. *Let $P = \nu x.P'$ Then $P\downarrow \iff P'\downarrow$ and $P\Downarrow \iff P'\Downarrow$.*

Proof. Let $\mathbb{D} := \nu x.[\cdot]$, It is easy to verify (since reduction is closed wrt. *PCtxt*-contexts) that for every process Q: $Q \xrightarrow{sr} Q'$ iff $\mathbb{D}[Q] \xrightarrow{sr} \mathbb{D}[Q']$, Q is successful iff $\mathbb{D}[Q]$ is successful, and $\mathbb{D}[Q] \xrightarrow{sr} Q''$ always implies $Q'' = \mathbb{D}[Q''']$ for some Q'''.

Corollary 4.18. *Let $P = \nu x_1, \ldots x_n.P'$ be in prenex-normal form such that P' contains no ν-binders. Then $P\downarrow \iff P'\downarrow$ and $P\Downarrow \iff P'\Downarrow$.*

Lemma 4.19. *For every process context $\mathbb{D}$ there exists a process context $\nu x_1.\ldots.\nu x_n.\mathbb{D}'$ where $\mathbb{D}'$ is ν-free and a variable-substitution σ such that for every process P: $\mathbb{D}[P] \equiv \nu x_1.\ldots.\nu x_n.\mathbb{D}'[\sigma(P)]$.*

Lemma 4.20 (Prenex Context Lemma). *For all well-formed processes P_1, P_2 it holds:*

- *If for all ν-free process contexts $\mathbb{D} \in PCtxt$ and all variable-substitutions σ: $\mathbb{D}[\sigma(P_1)]\downarrow \implies \mathbb{D}[\sigma(P_2)]\downarrow$, then $P_1 \leq_\downarrow P_2$.*
- *If for all ν-free process contexts $\mathbb{D} \in PCtxt$ and all variable-substitutions σ: $\mathbb{D}[\sigma(P_1)]\Downarrow \implies \mathbb{D}[\sigma(P_2)]\Downarrow$, then $P_1 \leq_\downarrow P_2$.*

Proof. We only show the first part for may-convergence, since the proof for should-convergence is completely analogous: Assume that for all ν-free process-contexts $\mathbb{D} \in PCtxt$ and all variable-substitutions σ: $\mathbb{D}[\sigma(P_1)]\downarrow \implies \mathbb{D}[\sigma(P_2)]\downarrow$. Now assume that there is a *PCtxt*-context $\mathbb{D}$ such that $\mathbb{D}[P_1]\downarrow$. We show that $\mathbb{D}[P_2]\downarrow$.Lemma 4.19 shows that $\mathbb{D}[P_1] \equiv \nu x_1, \ldots, x_n.\mathbb{D}_0[\sigma_0(P_1)]$ where $\mathbb{D}_0$ is a ν-free process context und σ is a variable substitution. By Proposition 4.6 we have $\nu x_1, \ldots, x_n.\mathbb{D}_0[\sigma_0(P_1)]\downarrow$. Corollary 4.18 shows that $\mathbb{D}_0[\sigma_0(P_1)]\downarrow$. Now the precondition of the lemma shows $\mathbb{D}_0[\sigma_0(P_2)]\downarrow$. Applying Corollary 4.18 again yields $\nu x_1, \ldots, x_n.\mathbb{D}_0[\sigma_0(P_2)]\downarrow$ and Lemma 4.19 shows $\nu x_1, \ldots, x_n.\mathbb{D}_0[\sigma_0(P_2)] \equiv \mathbb{D}[P_2]$. Finally, Proposition 4.6 implies $\mathbb{D}[P_2]\downarrow$.

4.4 Context Lemmas for Expressions

For the following lemmas we require expression *multicontexts*, i.e. contexts with several holes. We use $\widetilde{\mathbb{C}}$ for processes with several holes all on expression position. We write $\widetilde{\mathbb{C}}[\cdot_1, \ldots, \cdot_n]$ for such a multicontext ($\widetilde{\mathbb{C}}[\cdot_1^{\tau_1}, \ldots, \cdot_n^{\tau_n}]$ to make the types of the holes explicit) and $\widetilde{\mathbb{C}}[e_1, \ldots, e_n]$ for the process that results after replacing the holes by the expressions e_i where $e_i :: \tau_i$.

Remark 4.21. Let $\widetilde{\mathbb{C}}[\cdot_1^{\tau_1}, \ldots, \cdot_n^{\tau_n}]$ be a multicontext with n holes and $e_1 :: \tau_1, \ldots, e_n :: \tau_n$ be expressions such that $\widetilde{\mathbb{C}}[e_1, \ldots, e_n]$ is well-formed (not well-formed, resp.). Then for any expressions $e'_1 :: \tau_1, \ldots, e'_n :: \tau_n$ the process $\widetilde{\mathbb{C}}[e'_1, \ldots, e'_n]$ is well-formed (not well-formed, resp.). This holds, since the holes of $\widetilde{\mathbb{C}}$ are at expression position.

Lemma 4.22. *Let $\widetilde{\mathbb{C}}[\cdot_1^{\tau_1}, \ldots, \cdot_n^{\tau_n}]$ be a multicontext with n holes and for $i \in \{1, \ldots, n\}$ let $e_i :: \tau_i$ be expressions such that $\widetilde{\mathbb{C}}[e_1, \ldots, e_{i-1}, \cdot_i, e_{i+1}, \ldots, e_n]$ is a $\mathbb{D}[\mathbb{L}[\cdot]]$-context. Then there exists a hole $\cdot_j$ such that for arbitrary expressions $e'_1 :: \tau_1 \ldots e'_n :: \tau_n$ the context $\widetilde{\mathbb{C}}[e'_1, \ldots, e'_{j-1}, \cdot_j, e'_{j+1}, \ldots, e'_n]$ is a $\mathbb{D}[\mathbb{L}[\cdot]]$-context.*

Proof. Let the preconditions be met and $\widetilde{\mathbb{C}}[e_1, \ldots, e_{i-1}, \cdot_i, e_{i+1}, \ldots, e_n] = \mathbb{D}[\mathbb{L}[\cdot_i]]$ for some $\mathbb{D} \in PCtxt$ and $\mathbb{L} \in LCtxt$. We distinguish the cases:

- $\mathbb{L} = \mathbb{M}[\mathbb{F}]$. Then $\widetilde{\mathbb{C}}[e'_1, \ldots, e'_{i-1}, \cdot_i, e'_{i+1}, \ldots, e'_n]$ is always a context of the form $\mathbb{D}'[\mathbb{M}'[\mathbb{F}'[\cdot]]]$, since the holes $\cdot_1, \ldots, \cdot_{i-1}, \cdot_{i+1}, \ldots, \cdot \cdot_n$ of $\mathbb{C}$ cannot be on the path from the root to the hole $\cdot_i$ (where $\widetilde{\mathbb{C}}$ is seen as an expression tree).
- $\mathbb{L} = x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1$. If $\widetilde{\mathbb{C}}[e'_1, \ldots, e'_{i-1}, \cdot_i, e'_{i+1}, \ldots, e'_n]$ is not an $\mathbb{D}[\mathbb{L}[\cdot]]$-context, then at least one hole $\cdot_k \neq \cdot_i$ of $\widetilde{\mathbb{C}}$ must be on the path from the root to $\cdot_i$ in $\widetilde{\mathbb{C}}$. Moreover, this hole must be at one of the following positions:
  - $x \Leftarrow \mathbb{M}[\mathbb{F}[\cdot_k]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1$
  - $x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid \ldots \mid x_l = \mathbb{E}_l[\cdot_k] \mid \ldots \mid x_1 = \mathbb{E}_1$

  Let $\cdot_j$ be such a hole such that no other hole of $\cdot_1, \ldots, \cdot_n$ is on the path to $\cdot_j$. It is easy to see that for all expressions e'_i the context $\widetilde{\mathbb{C}}[e_1, \ldots, e_{j-1}, \cdot_j, e_{j+1}, \ldots, e_n]$ must also be an $\mathbb{D}[\mathbb{L}]$-context. □

Lemma 4.23. *For all variable permutations ρ:*

- *If for expressions e_1, e_2 and all $\mathbb{D} \in PCtxt$ and all $\mathbb{L} \in LCtxt$ it holds that $\mathbb{D}[\mathbb{L}[e_1]]\downarrow \implies \mathbb{D}[\mathbb{L}[e_2]]\downarrow$. Then for all $\mathbb{D} \in PCtxt$ and all $\mathbb{L} \in LCtxt$: $\mathbb{D}[\mathbb{L}[\rho(e_1)]]\downarrow \implies \mathbb{D}[\mathbb{L}[\rho(e_2)]]\downarrow$.*
- *If for expressions e_1, e_2 and all $\mathbb{D} \in PCtxt$ and all $\mathbb{L} \in LCtxt$ it holds that $\mathbb{D}[\mathbb{L}[e_1]]\uparrow \implies \mathbb{D}[\mathbb{L}[e_2]]\uparrow$. Then for all $\mathbb{D} \in PCtxt$ and all $\mathbb{L} \in LCtxt$: $\mathbb{D}[\mathbb{L}[\rho(e_1)]]\uparrow \implies \mathbb{D}[\mathbb{L}[\rho(e_2)]]\uparrow$.*

Lemma 4.24. *For all $n \geq 0$ and all multicontexts $\widetilde{\mathbb{C}}[\cdot_1^{\tau_1}, \ldots, \cdot_n^{\tau_n}]$ with n holes and for all expressions $e_1 :: \tau_1, \ldots, e_n :: \tau_n$ and $e'_1 :: \tau_1, \ldots, e'_n :: \tau_n$ holds:*

If for all $i = 1, \ldots, n$ and all $\mathbb{D}[\mathbb{L}[\cdot^{\tau_i}]]$-contexts: $\mathbb{D}[\mathbb{L}[e_i]]\downarrow \implies \mathbb{D}[\mathbb{L}[e'_i]]\downarrow$, then $\widetilde{\mathbb{C}}[e_1, \ldots, e_n]\downarrow \implies \widetilde{\mathbb{C}}[e'_1, \ldots, e'_n]\downarrow$

Proof. Let $\widetilde{\mathbb{C}}$, e_i, e'_i be given, such that the precondition holds for all e_i, e'_i and assume $\widetilde{\mathbb{C}}[e_1, \ldots, e_n]\downarrow$. We prove $\widetilde{\mathbb{C}}[e'_1, \ldots, e'_n]\downarrow$ by induction on the measure (l, n) (ordered lexicographically) where

- l is the length of a shortest standard reduction $\widetilde{\mathbb{C}}[e_1, \ldots, e_n] \xrightarrow{sr,l} Q$ where Q is successful.
- n is the number of holes of $\widetilde{\mathbb{C}}$.

The base case holds, since for $n = 0$ there is nothing to show. For the induction step we distinguish two cases:

- There is an index j such that $\widetilde{\mathbb{C}}[e_1,\ldots,e_{j-1},\cdot_j,e_{j+1},\ldots,e_n]$ is a $\mathbb{D}[\mathbb{L}]$-context. Then by Lemma 4.22 there is an index i, such that both $\widetilde{\mathbb{C}}[e_1,\ldots,e_{i-1},\cdot_i,e_{i+1},\ldots,e_n]$ and $\mathbb{C} := \widetilde{\mathbb{C}}[e'_1,\ldots,e'_{i-1},\cdot_i,e'_{i+1},\ldots,e'_n]$ are $\mathbb{D}[\mathbb{L}]$-contexts. Since the context $\widetilde{\mathbb{C}}[\cdot_1,\ldots,\cdot_{i-1},e_i,\cdot_{i+1},\ldots,\cdot_n]$ has $n-1$ holes the induction hypothesis implies $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_{i-1},e_i,e'_{i+1},\ldots,e'_n]\downarrow$. The precondition now shows that $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_{i-1},e'_i,e'_{i+1},\ldots,e'_n]\downarrow$ holds, since $\mathbb{C}$ is a $\mathbb{D}[\mathbb{L}]$-context.
- For no index j the context $\widetilde{\mathbb{C}}[e_1,\ldots,e_{j-1},\cdot_j,e_{j+1},\ldots,e_n]$ is a $\mathbb{D}[\mathbb{L}]$-context. If $\widetilde{\mathbb{C}}[e_1,\ldots,e_n]$ is successful then obviously $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_n]$ is successful, too. Thus let $l > 0$ and $\widetilde{\mathbb{C}}[e_1,\ldots,e_n] \xrightarrow{sr} Q$ be the first reduction step of a shortest reduction sequence for $\widetilde{\mathbb{C}}[e_1,\ldots,e_n]$ that ends in a successful process. This reduction can modify the context $\widetilde{\mathbb{C}}$, can remove or duplicate the expressions e_i, change the position of the expressions e_i and may rename the expressions e_i to establish the distinct variable convention. One can verify that Q must be of the form $\widetilde{\mathbb{C}}'[\rho(e_{\sigma(1)}),\ldots,\rho(e_{\sigma(m)})]$ such that $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_n] \xrightarrow{sr} \widetilde{\mathbb{C}}'[\rho'(e'_{\sigma(1)}),\ldots,\rho'(e'_{\sigma(m)})]$
  - ρ, ρ' are variable permutations of expressions
  - $\sigma : \{1,\ldots,m\} \to \{1,\ldots,n\}$,

  i.e. every modification, move, and remove done for $\widetilde{\mathbb{C}}[e_1,\ldots,e_n]$ can also be done for $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_n]$. Moreover, since α-renaming can be chosen arbitrarily we can replace the renamings ρ, ρ' by one renaming ρ'' such that $\widetilde{\mathbb{C}}[e_1,\ldots,e_n] \to \widetilde{\mathbb{C}}'[\rho''(e_{\sigma(1)}),\ldots,\rho''(e_{\sigma(m)})] =_\alpha Q$ and $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_n] \to \widetilde{\mathbb{C}}'[\rho''(e'_{\sigma(1)}),\ldots,\rho''(e'_{\sigma(m)})]$. From Lemma 4.23 it follows that for all $i \in \{1,\ldots,m\}$ and all $\mathbb{D} \in \mathit{PCtxt}$ and $\mathbb{L} \in \mathit{LCtxt}$ (with the the right type at the hole) the implication $\mathbb{D}[\mathbb{L}[\rho''(e_{\sigma(i)})]]\downarrow \implies \mathbb{D}[\mathbb{L}[\rho''(e'_{\sigma(i)})]]\downarrow$ holds. Since $\widetilde{\mathbb{C}}'[\rho''(e_{\sigma(1)}),\ldots,\rho''(e_{\sigma(m)})]$ has a shortest reduction sequence of length $l-1$ to a successful process we can apply the induction hypothesis and have $\widetilde{\mathbb{C}}'[\rho''(e'_{\sigma(1)}),\ldots,\rho''(e'_{\sigma(m)})]\downarrow$ and thus also $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_n]\downarrow$.

Lemma 4.25 (Context Lemma for May-Convergence). *$e_1 \leq_{\downarrow,\tau} e_2$ iff for all $\mathbb{D} \in \mathit{PCtxt}$ and $\mathbb{L}[\cdot^\tau] \in \mathit{LCtxt}$: $\mathbb{D}[\mathbb{L}[e_1]]\downarrow \implies \mathbb{D}[\mathbb{L}[e_2]]\downarrow$*

Proof. One direction is obvious, the other direction is a special case of Lemma 4.24 for $n = 1$.

Lemma 4.26. *For all $n \geq 0$ and all multicontexts $\widetilde{\mathbb{C}}[\cdot_1^{\tau_1},\ldots,\cdot_n^{\tau_n}]$ with n holes and for all expressions $e_1 :: \tau_1,\ldots,e_n :: \tau_n$ and $e'_1 :: \tau_1,\ldots,e'_n :: \tau_n$ holds:*

If for all $i = 1,\ldots,n$ and all $\mathbb{D}[\mathbb{L}[\cdot^{\tau_i}]]$-contexts: $\mathbb{D}[\mathbb{L}[e_i]]\uparrow \implies \mathbb{D}[\mathbb{L}[e'_i]]\uparrow$, and $\mathbb{D}[\mathbb{L}[e'_i]]\downarrow \implies \mathbb{D}[\mathbb{L}[e_i]]\downarrow$, then $\widetilde{\mathbb{C}}[e_1,\ldots,e_n]\uparrow \implies \widetilde{\mathbb{C}}[e'_1,\ldots,e'_n]\uparrow$

Proof. Let $\widetilde{\mathbb{C}}$, e_i, e'_i be given, such that the precondition holds for all e_i, e'_i and assume $\widetilde{\mathbb{C}}[e_1,\ldots,e_n]\uparrow$. We prove $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_n]\uparrow$ by induction on the measure (l,n) (ordered lexicographically) where

- l is the length of a shortest standard reduction $\widetilde{\mathbb{C}}[e_1,\ldots,e_n] \xrightarrow{sr,l} Q$ where $Q{\Uparrow}$.
- n is the number of holes of $\widetilde{\mathbb{C}}$.

The base case holds, since for $n = 0$ there is nothing to show. For the induction step we distinguish two cases:

- There is an index j such that $\widetilde{\mathbb{C}}[e_1,\ldots,e_{j-1},\cdot_j,e_{j+1},\ldots,e_n]$ is an $\mathbb{D}[\mathbb{L}]$-context. Then by Lemma 4.22 there is an index i, such that both $\widetilde{\mathbb{C}}[e_1,\ldots,e_{i-1},\cdot_i,e_{i+1},\ldots,e_n]$ and $\mathbb{C} := \widetilde{\mathbb{C}}[e'_1,\ldots,e'_{i-1},\cdot_i,e'_{i+1},\ldots,e'_n]$ are $\mathbb{D}[\mathbb{L}]$-contexts. Since the multicontext $\widetilde{\mathbb{C}}[\cdot_1,\ldots,\cdot_{i-1},e_i,\cdot_{i+1},\ldots,\cdot_n]$ has $n-1$ holes the induction hypothesis implies $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_{i-1},e_i,e'_{i+1},\ldots,e'_n]{\uparrow}$. The precondition now shows that $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_{i-1},e'_i,e'_{i+1},\ldots,e'_n]{\uparrow}$ holds, since $\mathbb{C}$ is an $\mathbb{D}[\mathbb{L}]$-context.
- There is no index j such that $\widetilde{\mathbb{C}}[e_1,\ldots,e_{j-1},\cdot_j,e_{j+1},\ldots,e_n]$ is an $\mathbb{D}[\mathbb{L}]$. If $\widetilde{\mathbb{C}}[e_1,\ldots,e_n]$ is must-divergent, then $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_n]$ is must-divergent: The precondition shows that for all $\mathbb{D}[\mathbb{L}]$-contexts: $\mathbb{D}[\mathbb{L}[e'_i]]{\downarrow} \implies \mathbb{D}[\mathbb{L}[e_i]]{\downarrow}$ and Lemma 4.24 shows $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_n]{\downarrow} \implies \widetilde{\mathbb{C}}[e_1,\ldots,e_n]{\downarrow}$. The last implication is equivalent to $\widetilde{\mathbb{C}}[e_1,\ldots,e_n]{\Uparrow} \implies \widetilde{\mathbb{C}}[e'_1,\ldots,e'_n]{\Uparrow}$.
  Now let $l > 0$ and $\widetilde{\mathbb{C}}[e_1,\ldots,e_n] \xrightarrow{sr} Q$ is the first reduction step of a shortest reduction sequence for $\widetilde{\mathbb{C}}[e_1,\ldots,e_n]$ that ends in a must-divergent process. This reduction can modify the context $\widetilde{\mathbb{C}}$, can remove or duplicate the expressions e_i, change the position of the expressions e_i and may rename the expressions e_i to establish the distinct variable convention. One can verify that Q must be of the form $\widetilde{\mathbb{C}}'[\rho(e_{\sigma(1)}),\ldots,\rho(e_{\sigma(m)})]$ such that $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_n] \xrightarrow{sr} \widetilde{\mathbb{C}}'[\rho'(e'_{\sigma(1)}),\ldots,\rho'(e'_{\sigma(m)})]$
  - ρ, ρ' are variable permutations of expressions
  - $\sigma : \{1,\ldots,m\} \to \{1,\ldots,n\}$,

  i.e. every modification, move, and remove done for $\widetilde{\mathbb{C}}[e_1,\ldots,e_n]$ can also be done for $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_n]$. Moreover, since α-renaming can be chosen arbitrarily we can replace the renamings ρ, ρ' by one renaming ρ'' such that $\widetilde{\mathbb{C}}[e_1,\ldots,e_n] \to \widetilde{\mathbb{C}}'[\rho''(e_{\sigma(1)}),\ldots,\rho''(e_{\sigma(m)})] =_\alpha Q$ and $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_n] \to \widetilde{\mathbb{C}}'[\rho''(e'_{\sigma(1)}),\ldots,\rho''(e'_{\sigma(m)})]$. From Lemma 4.23 it follows that for all $i \in \{1,\ldots,m\}$ and all $\mathbb{D} \in \mathit{PCtxt}$ and $\mathbb{L} \in \mathit{LCtxt}$ holds: $\mathbb{D}[\mathbb{L}[\rho''(e_{\sigma(i)})]]{\uparrow} \implies \mathbb{D}[\mathbb{L}[\rho''(e'_{\sigma(i)})]]{\uparrow}$. Since $\widetilde{\mathbb{C}}'[\rho''(e_{\sigma(1)}),\ldots,\rho''(e_{\sigma(m)})]$ has a shortest reduction sequence of length $l-1$ to a must-divergent process we can apply the induction hypothesis and have $\widetilde{\mathbb{C}}'[\rho''(e'_{\sigma(1)}),\ldots,\rho''(e'_{\sigma(m)})]{\uparrow}$ and thus also $\widetilde{\mathbb{C}}[e'_1,\ldots,e'_n]{\uparrow}$.

Lemma 4.27 (Context Lemma for Expressions). *Let e_1, e_2 be expressions of type τ such that for all $\mathbb{D} \in \mathit{PCtxt}$ and $\mathbb{L}[\cdot^\tau] \in \mathit{LCtxt}$: $\mathbb{D}[\mathbb{L}[e_1]]{\downarrow} \implies \mathbb{D}[\mathbb{L}[e_2]]{\downarrow}$ and $\mathbb{D}[\mathbb{L}[e_1]]{\Downarrow} \implies \mathbb{D}[\mathbb{L}[e_2]]{\Downarrow}$. Then $e_1 \leq_{c,\tau} e_2$.*

Proof. The inequation $e_1 \leq_{\downarrow,\tau} e_2$ follows from Lemma 4.25, and the inequation $e_1 \leq_{\Downarrow,\tau} e_2$ is an instance of Lemma 4.26.

5 Equivalences and Correct Program Transformations

A *program transformation* γ on processes is a binary relation on processes. It is called *correct* iff $\gamma \subseteq \sim_c$. A *program transformation* γ on expression is a binary relation on equally typed expressions. It is called *correct* iff $\gamma \subseteq \bigcup_\tau \sim_{c,\tau}$.

We show in this section that several transformations induced by standard reductions are correct program transformations, and also that several reduction rules are correct in any context.

We write (sr, a) (or alternatively $\xrightarrow{sr,a}$) to denote the standard reduction a. For a transformation γ we write $(\mathbb{D}, \gamma)$ (or alternatively $\xrightarrow{\mathbb{D},\gamma}$) to denote the closure of γ w.r.t. *PCtxt*-contexts, i.e. $(\mathbb{D}, \gamma) := \{(\mathbb{D}[P_1], \mathbb{D}[P_2]) \mid P_1 \xrightarrow{\gamma} P_2, \mathbb{D} \in \mathit{PCtxt}\}$. We use this notation also for other context classes, e.g. $(\mathbb{D}[\mathbb{L}], \gamma)$ is the closure of the transformation γ applied inside all *PCtxt*- and *LCtxt*-contexts.

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

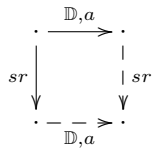
Remark 5.1. Without typing, the transformation $\texttt{return}\ e_1\ \texttt{>>=}\ e_2 \to (e_2\ e_1)$ would be incorrect. Consider the (untyped) context

$$\mathbb{C} := x \xleftarrow{\mathsf{main}} \texttt{case}_{Bool}\ [\cdot]\ \texttt{of}\ (\texttt{True} \to \texttt{True})\ (\texttt{False} \to \texttt{False}).$$

Then $\mathbb{C}[\texttt{return False >>= } \lambda x.\texttt{True}]\Uparrow$, but $\mathbb{C}[(\lambda x.\texttt{True})\ \texttt{False}]\Downarrow$.

Proposition 5.2. *The standard reductions* $(sr, lunit), (sr, nmvar), (sr, fork)$ *are correct transformations.*

Proof. Let $a \in \{(lunit), (nmvar), (fork)\}$. We show that (sr, a) does not change the convergence behavior in every *PCtxt*-context. If $\mathbb{D}[P] \xrightarrow{\mathbb{D},a} \mathbb{D}[Q]$ where P is the thread of the (a)-reduction, then the (a)-reduction is always also a standard reduction. Hence, $Q \leq_\downarrow P$ and $P \leq_\Downarrow Q$. For the remaining inequations we first observe that a $(\mathbb{D}, a)$-transformation always commutes with any standard reduction, if the reductions are different. I.e.:



This follows, since the $(\mathbb{D}, a)$-transformation is always a standard reduction and since it is deterministic for the corresponding thread and it does not require other components (like threads, MVars, bindings) for its applicability.

Now let $\mathbb{D}[P] \xrightarrow{\mathbb{D},a} \mathbb{D}[Q]$ and $\mathbb{D}[P]\downarrow$. By induction on the length of a successful standard reduction sequence for $\mathbb{D}[P]$ we show that there is a successful standard

reduction sequence for $\mathbb{D}[Q]$. If the sr-reduction and the $(\mathbb{D}, a)$-reduction are different, the induction step is covered by the above diagram. If the reductions are the same, then the we have a trivial case. For the induction base it clearly holds that if $\mathbb{D}[P]$ is successful, then $\mathbb{D}[Q]$ is successful, since $(\mathbb{D}, a)$ cannot be applicable to a terminated main thread. Thus $(a) \subseteq \leq_{\downarrow}$.

The remaining part is to show that $(a) \subseteq \geq_{\Downarrow}$. We show the analogous claim that if $\mathbb{D}[P]\uparrow$ then $\mathbb{D}[Q]\uparrow$. We use induction on a reduction sequence for $\mathbb{D}[P]$ that ends with a must-divergent process. The induction step follows by the diagram from above, the base case ($\mathbb{D}[P]\Uparrow \implies \mathbb{D}[Q]\uparrow$) holds, since we have already shown that $\mathbb{D}[Q]\downarrow \implies \mathbb{D}[P]\downarrow$, and $(\mathbb{D}, a)$ does not modify well-formedness of processes, thus $\mathbb{D}[P]\Uparrow \implies \mathbb{D}[Q]\Uparrow$.

Proposition 5.3. *The reduction $(unIO)$ is a correct transformation.*

Proof. Note that for well-formed and reducible processes, the (unIO)-reduction is always a standard-reduction. The same reasoning as for Proposition 5.2 applies, since it is sufficient to check the (unIO)-reduction in *PCtxt*-contexts.

The source of nondeterminism in CHF is the ability to concurrently access MVars from different threads. Hence, unsurprisingly the reduction rules for reading and writing into an MVar are not correct:

Proposition 5.4. *The reduction rules $(sr, tmvar)$ and $(sr, pmvar)$ are in general not correct.*

Proof. Consider the context

$$\mathbb{D} := x \overset{\mathsf{main}}{\Longleftarrow} \texttt{putMVar}\ y\ \texttt{True} \mid y\,\mathbf{m} - \mid [\cdot]$$

Then $\mathbb{D}[\mathbb{M}[\texttt{putMVar}\ y\ \texttt{True}]]$ may-converges:

$$\begin{array}{l} x \overset{\mathsf{main}}{\Longleftarrow} \texttt{putMVar}\ y\ \texttt{True} \mid y\,\mathbf{m} - \mid z \Leftarrow \mathbb{M}[\texttt{putMVar}\ y\ \texttt{True}] \\ \xrightarrow{sr} x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ () \mid y\,\mathbf{m}\,\texttt{True} \mid z \Leftarrow \mathbb{M}[\texttt{putMVar}\ y\ \texttt{True}] \end{array}$$

After the other $(sr, pmvar)$-reduction we get

$$\begin{array}{l} \mathbb{D}[z \Leftarrow \mathbb{M}[\texttt{putMVar}\ y\ \texttt{True}]] \\ \xrightarrow{sr,pmvar} x \overset{\mathsf{main}}{\Longleftarrow} \texttt{putMVar}\ y\ \texttt{True} \mid y\,\mathbf{m}\,\texttt{True} \mid z \Leftarrow \mathbb{M}[\texttt{return}\ ()] \end{array}$$

and the resulting process is must-divergent, since the `putMVar`-operation of the main thread is blocked indefinitely.

The counter example for (tmvar) is analogous, where all `putMVar`-operations are replaced by `takeMVar`-operations and the MVar y in the context $\mathbb{D}$ is filled.

Nevertheless, if the execution of a $(sr, tmvar)$- or $(sr, pmvar)$-reduction is deterministic, it is a correct program transformation. For formalizing this we define further transformations related to reduction rules (we also add a rule for garbage collection):

Definition 5.5. *The transformations (dtmvar), (dpmvar), and (gc) are defined as follows, where we assume the transformations to be closed w.r.t. structural congruence and w.r.t. PCtxt.*

(dtmvar) $\nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{takeMVar}\ x] \mid x\,\mathbf{m}\,e] \ \rightarrow\ \nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{return}\ e] \mid x\,\mathbf{m}\,-]$

if for all $\mathbb{D}' \in$ PCtxt and all $\xrightarrow{sr,*}$-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) $\nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{putMVar}\ x\ e] \mid x\,\mathbf{m}\,-] \ \rightarrow\ \nu x.\mathbb{D}[y \Leftarrow \mathbb{M}[\texttt{return}\ ()] \mid x\,\mathbf{m}\,e]$

if for all $\mathbb{D}' \in$ PCtxt and all $\xrightarrow{sr,*}$-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

(gc) $\nu x_1,\ldots,x_n.(P \mid \texttt{Comp}(x_1) \mid \ldots \mid \texttt{Comp}(x_n)) \ \rightarrow\ P$
if for all $i \in \{1,\ldots,n\}$: $\texttt{Comp}(x_i)$ is either a binding $x_i = e_i$, an MVar $x_i\,\mathbf{m}\,e_i$, an empty MVar $x_i\,\mathbf{m}\,-$, and $x_1,\ldots,x_n$ do not occur as free variables in P.

Remark 5.6. Note that 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`.

However, note that the reduction may be able to construct a disturbing execution of ($\texttt{takeMVar}\ x$) or ($\texttt{putMVar}\ x\ e'$) also in non-obvious cases. For instance, consider the process $P := \nu x.(y \Leftarrow \texttt{takeMVar}\ x \mid x\,\mathbf{m}\,e \mid z\,\mathbf{m}\,x)$. It does not fulfill the criteria of (dtmvar), since e.g. for the context

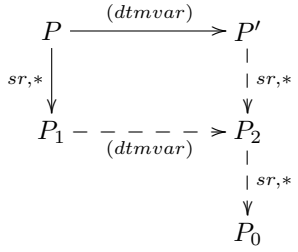
$$\mathbb{D}' = y' \Leftarrow \texttt{takeMVar}\ z \texttt{ >>= } \lambda z'.\texttt{takeMVar}\ z' \mid [\cdot]$$

the MVar x can be accessed in $\mathbb{D}'[P]$ by the y'-thread *before* the ($\texttt{takeMVar}\ x$)-operation of the y-thread is executed.

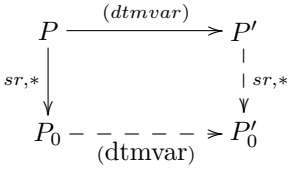
Proposition 5.7. *The transformations (gc), (dtmvar), and (dpmvar) are correct program transformations.*

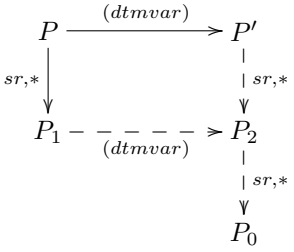
Proof. For (gc), this is obvious, since (gc) is not among the standard reductions, and there is no interaction with standard reductions. Now consider the transformation (dtmvar), which is also a standard reduction. Let $P \xrightarrow{(dtmvar)} P'$. If $P'\downarrow$, then obviously $P\downarrow$, since (dtmvar) is a standard reduction. If $P\downarrow$, then $P \xrightarrow{sr,*} P_0$, where P_0 is successful. The precondition for the application of (dtmvar), that no takeMVar can interfere during the reduction, shows that only a square forking diagram holds, and thus either the same reduction can be used

for P', or the reduction is of the form:



Hence also $P'{\downarrow}$.
We have also proved that $P{\Uparrow} \iff P'{\Uparrow}$. If $P'{\uparrow}$, then we also have $P{\uparrow}$, since $P \xrightarrow{(dtmvar)} P'$ is a standard reduction and by the previous equivalence.
Now assume $P{\uparrow}$. I.e. $P \xrightarrow{sr,*} P_0$ with $P_0{\Uparrow}$. The same argument on forking shows that one of the following constructions of reduction holds:





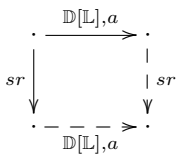
In the first case, also $P'_0{\Uparrow}$, since $\xrightarrow{(dtmvar)}$ is a standard reduction, and in the second case, P_0 is already must-divergent.

The arguments for (dpmvar) are completely analogous.

In the following we speak of the transformations (lbeta), (case), (seq), (mkbinds), and mean the reduction without the $\mathbb{L}$-context, i.e. only the modification of the redex.

Proposition 5.8. *The transformations (lbeta), (case), (seq), (mkbinds) are correct as transformation in any context.*

Proof. Let e_1, e_2 be equally typed expressions, such that $e_1 \xrightarrow{a} e_2$. Let $\mathbb{D}[\mathbb{L}[e_1]] \xrightarrow{\mathbb{D}[\mathbb{L}],a} \mathbb{D}[\mathbb{L}[e_2]]$ for $a \in \{(\text{lbeta}), (\text{case}), (\text{seq}), (\text{mkbinds})\}$ where e_1 is the redex of the a-reduction. Then the a-reduction is always also a standard reduction. Hence, by the context lemma, $e_2 \leq_{\downarrow} e_1$. To show the remaining inequation $e_1 \leq_{\downarrow} e_2$, we first observe that a $(\mathbb{D}[\mathbb{L}], a)$-transformation always commutes with any standard reduction, if the reductions are different. I.e.:



This follows, since the $(\mathbb{D}[\mathbb{L}[\cdot]], a)$-transformation is always a standard reduction and since it is deterministic for the corresponding thread. Now let $\mathbb{D}[\mathbb{L}[e_1]] \xrightarrow{\mathbb{D}[\mathbb{L}],a} \mathbb{D}[\mathbb{L}[e_2]]$ and $\mathbb{D}[\mathbb{L}[e_1]]\downarrow$. By induction on the length of a successful standard reduction sequence for $\mathbb{D}[\mathbb{L}[e_1]]$ we show that there is a successful standard reduction sequence for $\mathbb{D}[\mathbb{L}[e_2]]$. The induction step is covered by the above diagram. For the induction base it clearly holds that if $\mathbb{D}[\mathbb{L}[e_1]]$ is successful, then $\mathbb{D}[\mathbb{L}[e_2]]$ is successful, since $(\mathbb{D}[\mathbb{L}], a)$ cannot be applicable to the terminated main thread. Thus the context lemma for may-convergence shows $(a) \subseteq \leq_{\downarrow}$, and hence $(a) \subseteq \sim_{\downarrow}$.

From $\mathbb{D}[\mathbb{L}[e_1]] \xrightarrow{\mathbb{D}[\mathbb{L}],a} \mathbb{D}[\mathbb{L}[e_1]]$ and $(a) \subseteq \sim_{\downarrow}$, we now derive, using the context lemma, that $e_1 \leq e_2$.

To show the remaining inequation $e_2 \leq_{\Downarrow} e_1$ we have to show that $(\mathbb{D}[\mathbb{L}], a) \subseteq \geq_{\Downarrow}$. We show the equivalent claim that if $\mathbb{D}[\mathbb{L}[e_1]]\uparrow$ then $\mathbb{D}[\mathbb{L}[e_2]]\uparrow$. Therefore we use induction on a reduction sequence for $\mathbb{D}[\mathbb{L}[e_1]]$ that ends with a must-divergent process. The induction step follows by the diagram from above, the base case $(\mathbb{D}[\mathbb{L}[e_1]]\Uparrow \implies \mathbb{D}[\mathbb{L}[e_2]]\uparrow)$ holds, since we have already shown that $\mathbb{D}[\mathbb{L}[e_2]]\downarrow \implies \mathbb{D}[\mathbb{L}[e_1]]\downarrow$ and thus $\mathbb{D}[\mathbb{L}[e_1]]\Uparrow \implies \mathbb{D}[\mathbb{L}[e_2]]\Uparrow$. Thus the context lemma implies the last part $e_2 \leq_{\Downarrow} e_1$.

It remains to show correctness of the copy reductions (*cp*) and (*cpcx*) (i.e. the reduction rules (sr, cp) and $(sr, cpcx)$ where the $\widehat{\mathbb{L}}$ is replaced by an arbitrary process context $\mathbb{C}$ with an expression hole). The correctness proof is not straightforward and thus requires further proof techniques. We will show the correctness in the next section using infinite trees.

6 Correctness of Call-by-Name Reductions

This section contains proofs for the correctness of call-by-name reductions, and the equivalence of call-by-need and call-by-name evaluation.

The main technique for the proof is to use infinite terms and the corresponding reductions, which allows one to encode recursive bindings into expressions. This technique was used in [SS07] to show correctness of inlining in the deterministic call-by-need lambda calculus with letrec and also in [SSSM10] to show equivalence of the call-by-need lambda calculus with letrec and the lazy lambda calculus [Abr90].

6.1 Calculus for Infinite Trees

We define infinite expressions which are intended to be the letrec- and binding-unfolding of the expression with the extra condition that cyclic variable chains lead to local nontermination.

Definition 6.1. Infinite expressions *IExpr are defined like expressions Expr omitting the letrec-component, adding a constant* `Bot`*, and interpreting the gram-*

mar coinductively, i.e. the grammar is as follows

$$s,t,s_i,t_i \in \mathit{IExpr} ::= x \mid ms \mid \texttt{Bot} \mid \lambda x.s \mid (s_1\ s_2) \mid c\ s_1 \ldots s_{\mathrm{ar}(c)} \mid \texttt{seq}\ s_1\ s_2 \\ \mid \texttt{case}_T\ s\ \texttt{of}\ (c_{T,1}\ x_1 \ldots x_{\mathrm{ar}(c_{T,1})} \to s_1) \ldots (c_{T,|T|}\ x_1 \ldots x_{\mathrm{ar}(c_{T,|T|})} \to s_{|T|})$$

$$ms \in \mathit{IMExpr} ::= \texttt{return}\ s \mid s_1\ \texttt{>>=}\ s_2 \mid \texttt{forkIO}\ s \\ \mid \texttt{takeMVar}\ s \mid \texttt{newMVar}\ s \mid \texttt{putMVar}\ s_1\ s_2$$

Infinite processes *(or* tree processes*) IProc are defined like usual processes Proc using the same (inductive) grammar omitting bindings, with an additional process* $\mathbf{0}$*, and infinite expressions instead of expressions. I.e. the grammar is:*

$$S,T,S_i,T_i \in \mathit{IProc} ::= S_1 \mid S_2 \mid x \Leftarrow s \mid \nu x.S \mid x\,\mathbf{m}\,s \mid x\,\mathbf{m}- \mid \mathbf{0}$$

The process $\mathbf{0}$ *is like a process without any reduction rules. Structural congruence on tree processes is defined as for processes where we add the congruence equation* $\mathbf{0} \mid S \equiv S$*.*

Thus there are finitely many process components, but perhaps infinite expressions in threads or MVars. In order to distinguish the following the usual processes and expressions from the infinite ones, we say *tree* or infinite expressions or tree process or infinite process in order to distinguish the usual notions from the ones for infinite processes. In infinite processes there are no variables for bindings, but variables corresponding to threads or MVars are there, and remain as free variables within the infinite expressions. The constant `Bot` in expressions is without any reduction rule. It will represent cyclic bindings that are only via variables like $x = y \mid y = x$.

In the following explicit definition of a mapping from processes to their infinite image, we sometimes use the explicit binary application operator @ for applications inside the trees (i.e. an application in the tree is sometimes written as $(@\ s_1\ s_2)$ instead of $(s_1\ s_2)$), since it is easier to explain, but stick to the common notation in examples.

Definition 6.2. *Let* P *be a process. The translation* $IT :: Proc \to IProc$ *translates a process* P *into its infinite tree process* $IT(P)$*. Instead of providing a direct definition of the mapping* IT*, we provide an algorithm that given a position* p *of the infinite tree and a given process* P *it computes the label of* $IT(P)$ *at position* p*. A position is a sequence of positive integers, where the empty position is denoted as* ε*. We use Dewey notation for positions, i.e. the position* $i.p$ *is the sequence starting with* i *followed by position* p*. The computation starts with* $P|_p$ *and then proceeds with the rules given in Figure 4. The first rules define the computed label for the position* ε*, the second part of the rules describes the general case for positions. If the computation fails (or is undefined), then the position is not valid in the tree.*

The equivalence of infinite processes is syntactic, where α*-equal trees are assumed to be equivalent.*

$$
\begin{array}{lll}
\mathbb{D}[(x \Leftarrow e)|_{\varepsilon}] & \mapsto x \Leftarrow & \\
\mathbb{D}[(x\,\mathbf{m}\,e)|_{\varepsilon}] & \mapsto x\,\mathbf{m} & \\
\mathbb{D}[(x\,\mathbf{m}\,-)|_{\varepsilon}] & \mapsto x\,\mathbf{m}\,- & \\
\mathbb{D}[(\nu x.Q)|_{\varepsilon}] & \mapsto \nu x & \\
\mathbb{D}[(Q_1 \mid Q_2)|_{\varepsilon}] & \mapsto \mid & \\
\mathbb{D}[(x = e)|_{\varepsilon}] & \mapsto \mathbf{0} & \\
\mathbb{C}[(e_1\ e_2)|_{\varepsilon}] & \mapsto @ & \\
\mathbb{C}[(\mathtt{case}_T\ \ldots)|_{\varepsilon}] & \mapsto \mathtt{case}_T & \\
\mathbb{C}[(c\ x_1 \ldots\ x_n \to e)|_{\varepsilon}] & \mapsto (c\ x_1\ \ldots\ x_n) & \text{for a case-alternative} \\
\mathbb{C}[(\mathtt{seq}\ e_1\ e_2)|_{\varepsilon}] & \mapsto \mathtt{seq} & \\
\mathbb{C}[(c\ e_1 \ldots e_n)|_{\varepsilon}] & \mapsto c & \\
\mathbb{C}[(\lambda x.e)|_{\varepsilon}] & \mapsto \lambda x & \\
\mathbb{C}[x|_{\varepsilon}] & \mapsto x & \text{if } x \text{ is a free variable, a thread-identifier,} \\
 & & \text{a name of an MVar, a } \nu\text{-bound variable,} \\
 & & \text{or a lambda-bound variable in } \mathbb{C}[x]
\end{array}
$$

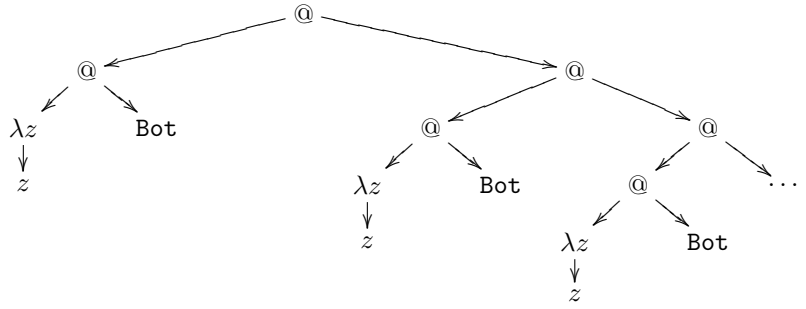
If the position ε hits the same (let- or binding-bound) variable twice, using the rules below, then the result is `Bot`. The general case is:

$$
\begin{array}{ll}
\mathbb{D}[(x \Leftarrow e)|_{1.p}] & \mapsto \mathbb{D}[(x \Leftarrow e|_p)] \\
\mathbb{D}[(x\,\mathbf{m}\,e)|_{1.p}] & \mapsto \mathbb{D}[(x\,\mathbf{m}\,e|_p)] \\
\mathbb{D}[(\nu x.Q)|_{1.p}] & \mapsto \mathbb{D}[(\nu x.Q|_p)] \\
\mathbb{D}[(Q_1 \mid Q_2)|_{1.p}] & \mapsto \mathbb{D}[(Q_1|_p \mid Q_2)] \\
\mathbb{D}[(Q_1 \mid Q_2)|_{2.p}] & \mapsto \mathbb{D}[(Q_1 \mid Q_2|_p)] \\
\mathbb{C}[(\lambda x.e)|_{1.p}] & \mapsto \mathbb{C}[\lambda x.(e|_p)] \\
\mathbb{C}[(e_1\ e_2)|_{1.p}] & \mapsto \mathbb{C}[(e_1|_p\ e_2)] \\
\mathbb{C}[(e_1\ e_2)|_{2.p}] & \mapsto \mathbb{C}[(e_1\ e_2|_p)] \\
\mathbb{C}[(\mathtt{seq}\ e_1\ e_2)|_{1.p}] & \mapsto \mathbb{C}[(\mathtt{seq}\ e_1|_p\ e_2)] \\
\mathbb{C}[(\mathtt{seq}\ e_1\ e_2)|_{2.p}] & \mapsto \mathbb{C}[(\mathtt{seq}\ e_1\ e_2|_p)] \\
\mathbb{C}[(\mathtt{case}\ e\ \mathtt{of}\ alt_1 \ldots alt_n)|_{1.p}] & \mapsto \mathbb{C}[(\mathtt{case}\ e|_p\ \mathtt{of}\ alt_1 \ldots alt_n)] \\
\mathbb{C}[(\mathtt{case}\ e\ \mathtt{of}\ alt_1 \ldots alt_n)|_{(i+1).p}] & \mapsto \mathbb{C}[(\mathtt{case}\ e\ \mathtt{of} alt_1 \ldots alt_i|_p \ldots alt_n)] \\
\mathbb{C}[\ldots(c\ x_1\ \ldots\ x_n \to e)|_{1.p}\ldots] & \mapsto \mathbb{C}[\ldots(c\ x_1\ \ldots\ x_n \to e|_{1.p})\ldots] \\
\mathbb{C}[(c\ e_1 \ldots e_n)|_{i.p}] & \mapsto \mathbb{C}[(c\ e_1 \ldots e_i|_p \ldots e_n)] \\
 & \text{where } c \text{ is a constructor or a monadic constant} \\
\mathbb{C}[(\mathtt{letrec}\ Env\ \mathtt{in}\ e)|_p] & \mapsto \mathbb{C}[(\mathtt{letrec}\ Env\ \mathtt{in}\ e|_p)] \\
\mathbb{C}_1[(\mathtt{letrec}\ x = e, Env\ \mathtt{in}\ \mathbb{C}_2[x|_p])] & \mapsto \mathbb{C}_1[(\mathtt{letrec}\ x = e|_p, Env\ \mathtt{in}\ \mathbb{C}_2[x])] \\
\mathbb{C}_1[(\mathtt{letrec}\ x = e_1, y = \mathbb{C}_2[x|_p], Env\ \mathtt{in}\ e_2)] & \\
 & \mapsto \mathbb{C}_1[(\mathtt{letrec}\ x = e_1|_p, y = \mathbb{C}_2[x], Env\ \mathtt{in}\ e_2)] \\
\widetilde{\mathbb{D}}[x = e, \mathbb{C}[x|_p]] & \mapsto \widetilde{\mathbb{D}}[x = e|_p, \mathbb{C}[x]]
\end{array}
$$

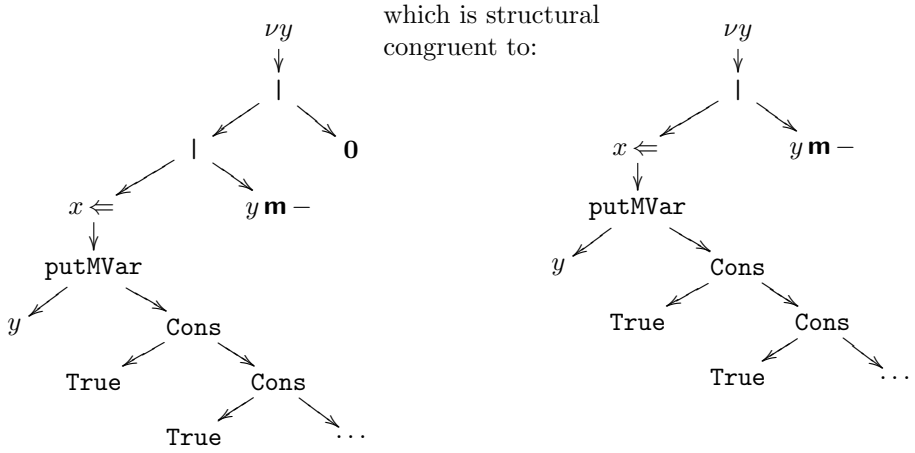
where the notation $\widetilde{\mathbb{D}}[\cdot,\cdot]$ means a two-hole process context with process holes, i.e. a process where two subprocesses are replaced by context holes.

Fig. 4. Infinite tree construction from positions for fixed P

Example 6.3. The expression `letrec` $x = x, y = (\lambda z.z)\ x\ y$ `in` y has the corresponding tree $((\lambda z.z)$ `Bot` $((\lambda z.z)$ `Bot` $((\lambda z.z)$ `Bot` $\ldots)))$ or written explicitly:



Example 6.4. For the process $\nu y.((x \Leftarrow$ `putMVar` $y\ z \mid y\,\mathbf{m}-) \mid z =$ `Cons True` $z)$ the corresponding infinite tree is:



which is structural congruent to:

Similar as for processes we can use the prenex-normal form w.r.t. ν for infinite processes.

We use different classes of contexts for infinite processes and trees. In abuse of notation we use the same symbols for these contexts as for the contexts defined previously on (finite) processes and expressions.

Definition 6.5. *Process contexts IProc, call-by-name evaluation contexts IECtxt, forcing contexts IFCtxt, and monadic contexts IMCtxt are defined as follows where all grammars are interpreted inductively and where $S \in$ IProc, $s \in$ IExpr:*

$$\begin{aligned}
\mathbb{D}, \mathbb{D}_i \in \mathit{IPCtxt} &::= [\cdot] \mid \mathbb{D} \mid S \mid S \mid \mathbb{D} \mid \nu x.\mathbb{D} \\
\mathbb{M}, \mathbb{M}_i \in \mathit{IMCtxt} &::= [\cdot] \mid \mathbb{M} \texttt{ >>= } s \\
\mathbb{F}, \mathbb{F}_i \in \mathit{IFCtxt} &::= \mathbb{E} \mid (\texttt{takeMVar}\ \mathbb{E}) \mid (\texttt{putMVar}\ \mathbb{E}\ s) \\
\mathbb{E}, \mathbb{E}_i \in \mathit{IECtxt} &::= [\cdot] \mid (\mathbb{E}\ s) \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ \mathit{alts}) \mid (\texttt{seq}\ \mathbb{E}\ s)
\end{aligned}$$

A reduction context $\mathbb{R} \in$ *IRCtxts for infinite processes is constructed as* $\mathbb{D}[V \Leftarrow \mathbb{M}[\mathbb{F}]]$ *for* $\mathbb{D} \in$ *IPCtxt,* $\mathbb{M} \in$ *IMCtxt, and* $\mathbb{F} \in$ *IFCtxt.*

Lemma 6.6. *Let* P, Q *be processes and* $P \to Q$ *by a rule (cp), (cpcx) or (mkbinds). Then* $IT(P) = IT(Q)$.

Definition 6.7. *The functional reduction rules on tree processes are allowed in any context and are as follows:*

(betaTr) $((\lambda x.s)\ r) \to s[r/x]$
(seqTr) $(\texttt{seq}\ s\ t) \quad \to \quad t \quad$ *if* s *is a functional value*
(caseTr) $(\texttt{case}_T\ (c\ s_1 \ldots s_n)\ \texttt{of}\ \ldots (c\ x_1 \ldots x_n) \to s \quad \to \quad s[s_1/x_1, \ldots, s_n/x_n]$

The monadic computation rules are unchanged for (lunit), (tmvar), (pmvar), (nmvar), and (forkIO) (see Fig. 3 where now $\mathbb{M}$ *denotes an IMCtxt-context) and adapted for the rule (unIO):*

(unIOTr) $\mathbb{D}[y \Leftarrow \texttt{return}\ y] \xrightarrow{SR} (\mathbb{D}[\mathbf{0}])[\texttt{Bot}/y]$

(unIOTr) $\mathbb{D}[y \Leftarrow \texttt{return}\ s] \xrightarrow{SR} (\mathbb{D}[\mathbf{0}])[s//y]$
if $s \neq y$*; and the thread is not the main-thread*
where // means the infinite recursive replacement of s *for* y*;*
and where $\mathbb{D}$ *means the whole process that is in scope of* y.

If a tree-process-reduction rule (betaTr), (caseTr), or (seqTr) is applied within an IRCtxts-context, or it is a mondaic rule, then we call it a standard-reduction (SR-reduction) on tree processes, and write $T \xrightarrow{SR} T'$ *A successful infinite process is an infinite process where the main thread exists and is of the form* $y \Leftarrow \texttt{return}\ e$*. We also use the convergence predicates* $\downarrow, \Uparrow, \Downarrow, \uparrow$ *for infinite tree processes, which are defined accordingly. The* redex *of a tree process reduction is the (infinite) subtree which is modified by the reduction rule. Note that for reduction rule (unIOTr) the redex is the whole infinite tree.*

Note that $\xrightarrow{betaTr,SR}$ and $\xrightarrow{caseTr,SR}$ only reduce a single redex, but may modify infinitely many positions, since there may be infinitely many positions of the replaced variable x. E.g. a (SR,betaTr) of $IT((\lambda x.(\texttt{letrec}\ z = (z\ x)\ \texttt{in}\ z))\ r) = (\lambda x.((\ldots\ (\ldots\ x)\ x)\ x))\ r \to\ ((\ldots\ (\ldots\ r)\ r)\ r))$ replaces the infinite number of occurrences of x by r.

Lemma 6.8. *Let* P *be a process. If* P *is successful then* $IT(P)$ *is successful. If* $IT(P)$ *is successful, then* $P\downarrow$.

Proof. If P is successful, then obviously, $IT(P)$ is successful.
If $IT(P)$ is successful, then in P the main thread may be $y \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ s$, and the claim holds. The main thread may also be $y \overset{\mathsf{main}}{\Longleftarrow} x$, where x is bound via several bindings to ($\texttt{return}\ s$), perhaps decorated with `letrec`-environments. We

claim that there exists a sequence of sr-reductions $P \xrightarrow{(cp)\vee(cpcx)\vee(mkbinds),*,sr} P'$, where P' is is successful. Since IT computes an infinite expression of the form (`return` s) at the position of x, there is a chain of variables of length n for x as follows: starting with x where all intermediate bindings are either $x_i = \mathbb{LR}_i[x_{i+1}]$ in a process binding or a letrec-binding, and the last one is $x_n = \mathbb{LR}_n[(\texttt{return}\ s)]$, and the contexts $\mathbb{LR}_i$ are contexts according to the grammar $\mathbb{LR} ::= [\cdot] \mid (\texttt{letrec}\ Env\ \texttt{in}\ \mathbb{LR})$. There is a sequence of sr-reductions of P using (cp), ($cpcx$), and ($mkbinds$) always modifying either $y \overset{\mathsf{main}}{\Longleftarrow} x$ by copying a variable for x, or performing a ($mkbinds$) on $y \overset{\mathsf{main}}{\Longleftarrow} x \mid x = (\texttt{letrec}\ Env\ \texttt{in}\ t)$, or as the final reduction, a (cpcx) is performed. The sr-reduction sequence terminates with a successful process, since it strictly decreases the label-computing steps for position of x. Note that the reductions do not change the infinite process due to Lemma 6.6.

We will use a variant of infinite outside-in developments [Bar84,KKSdV97] as a reduction on trees that may reduce infinitely many redexes in one step.

Definition 6.9. *We define an infinite variant of Barendregt's 1-reduction: Let $T \in IProc$ be an infinite process. Let M be a set of (perhaps infinitely many) labelled redexes of T. We require that the set M*

- *contains only redexes of the same reduction rule*
- *is a singleton, whenever the reduction rule is different from (betaTr), (seqTr), or (caseTr)*

Then by $T \xrightarrow{I,M} T'$ we denote the (perhaps infinite) development top down, i.e. the reduction sequence constructs a new infinite tree T' top-down by using labelled reduction for every labelled redex, where the label of the redex is removed before the reduction. If the reduction does not terminate for a subtree at the top level of the subtree, then this subtree is replaced by the constant `Bot` *in the result T'. If the subtree is of the form $(c\ s_1 \ldots s_n)$ where c stands for any syntactic construct, and no superexpression carries a reduction label, then for all i let s'_i be the resulting infinite tree for the infinite development of s_i. Then the resulting tree is $(c\ s'_1 \ldots s'_n)$. This recursively defines the resulting tree top-down.*

If the reduction $T \xrightarrow{I,M} T'$ does not contain standard redexes, then we write $T \xrightarrow{I,M,NSR} T'$. We write $T \xrightarrow{I,NSR} T'$ ($T \xrightarrow{I} T'$, resp.) if there exists a set M such that $T \xrightarrow{I,M,NSR} T'$ ($T \xrightarrow{I,M} T'$, resp.).

Example 6.10. We give two examples of corresponding infinite standard reductions.
An sr-reduction of a process corresponds to an $\xrightarrow{I,M}$-reduction and maybe corresponds to an infinite sequence of infinite SR-reductions. Consider $z \Leftarrow y \mid y = (\lambda x.y)\ a$. The (sr,lbeta)-reduction results in $z \Leftarrow y \mid y = y \mid x = a$. The corresponding infinite process will be $z \Leftarrow (\lambda x.(\lambda x.(\ldots)\ a))\ a$, and the ($SR$,betaTr)-reduction-sequence is infinite.

Let the expression be $z \Leftarrow y \mid y = (\texttt{seq}\ c\ (\texttt{seq}\ y\ d)$, where c, d are constructor constants. Then the standard reduction results in $z \Leftarrow y \mid y = (\texttt{seq}\ y\ d)$, which diverges. The corresponding infinite process is $z \Leftarrow (\texttt{seq}\ c\ (\texttt{seq}\ ((\texttt{seq}\ c\ (\texttt{seq}\ (\ldots)\ d))\ d))$, which has an infinite number of *SR*-reductions, at an infinite number of deeper and deeper positions.

6.2 Equivalence of Tree-Convergence and Process-Convergence

In this section we will show that (may- and should-) convergence for processes P coincides with (may- and should-) convergence for the corresponding infinite tree $IT(P)$. We will consider may-convergence, and thereafter we consider should-convergence. Some easy facts are shown in the following two lemmas:

Lemma 6.11. *Let P be a process such that $P \xrightarrow{sr,a} P'$. If the reduction is monadic, then $IT(P) \xrightarrow{SR,a} IT(P')$ If the reduction a is (cp), (cpcx) or (mkbinds) then $IT(P) = IT(P')$. If the reduction a is (lbeta), (case), or (seq) then $IT(P) \xrightarrow{I,M,a'} IT(P')$ for some M, where a' is (lbetaTr), (caseTr), or (seqTr), respectively, some M, and the set M contains standard redexes.*

Proof. Only the latter needs a justification. Therefore, we label every redex in $IT(P)$ that is derived from the redex $P \xrightarrow{sr} P'$ by $IT(.)$. This results in the set M for $IT(P)$. There will be at least one position in M that is a standard redex of $IT(P)$.

Lemma 6.12. *Let T be an infinite process. If $T \xrightarrow{I,NSR} T'$, where T' is successful. Then also T is successful.*

Proof. The successful thread $y \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ s$ cannot be generated by $\xrightarrow{I,NSR}$-reductions: at least one reduction must be at a standard reduction position, which would be a standard reduction.

6.2.1 Standardization of Tree Reduction for May-Convergence. In this subsection we show that for an arbitrary reduction sequence on infinite trees resulting in a successful process we can construct an *SR*-reduction sequence that also results in a successful process. We prove this result in a series of lemma.

Definition 6.13. *For tree processes we use the following notation for positions. For a finite or infinite process T a position p where $T|p$ is an expression can be split into a prefix p_0 and a suffix p_1 such that, $p = p_0.p_1$ and p_0 is the position of the top-level expression of a process component, i.e. p_0 is the position of the expression of the top-level expression of a concurrent thread, or of the expression which is the content of a filled MVar. For empty MVars the suffix p_1 is the empty string.*

Example 6.14. We consider the process

$$P = \nu x_1.\nu x_2.((x_1 \Leftarrow \texttt{putMVar}\ y\ z \mid (y\,\mathbf{m} - \mid x_2\,\mathbf{m}\,\lambda w.w)) \mid z = \texttt{Cons True}\ z).$$

The infinite tree $IT(P)$ in prenex normal form with removed $\mathbf{0}$ is structural congruent to

$$IT(P) \equiv \nu x_1.\nu x_2.((x_1 \Leftarrow \texttt{putMVar}\ y\ (\texttt{Cons True}\ \ldots)) \mid (y\,\mathbf{m} - \mid x_2\,\mathbf{m}\,\lambda w.w))).$$

The top level expression positions are 1.1.1.1 for the top-level expression $\texttt{putMVar}\ y\ (\texttt{Cons True}\ \ldots)$ in the thread $x_1 \Leftarrow \texttt{putMVar}\ y\ (\texttt{Cons True}\ \ldots))$ and 1.1.2.2.1 for the expression $\lambda w.w$ in the filled MVar $x_2\,\mathbf{m}\,\lambda w.w$.

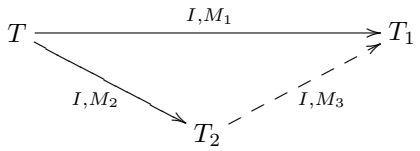
Consider a reduction $T \xrightarrow{I,M} T'$ of type (betaTr), (caseTr) or (seqTr). This reduction may have an *SR*-component and can be split into $T \xrightarrow{SR} T_1 \xrightarrow{I} T'$. This can be iterated, as long as the remaining $T_1 \xrightarrow{I} T'$ has an *SR*-component. Unfortunately, this split process may be non-terminating. Since this split is essential for our proof technique, we have to introduce a new notion and notation: We split the reduction into the parts for the different process components: We partition the set $M = \bigcup_{i=1,\ldots,n} W_i$, where all positions of a set W_i have cp_i as prefix and where cp_i is the position of the top level expression of the component with index i. This is possible since this split is only necessary for the reduction rules (betaTr), (caseTr) and (seqTr), and since these reductions only modify a single process component. If the split results for a component j in an infinite *SR*-reduction sequence then we write the reduction as $\xrightarrow{I,W_j,infSR}$. Since the reductions are parallel, we can join all the sets and obtain a partial reduction $\xrightarrow{I,M_{inf},infSR}$ where M_{inf} contains all positions in process components, where the corresponding *SR*-reduction sequence that follows from M is infinite. Let $M_{fin} = M \setminus M_{inf}$, i.e. the other labeled positions. This reduction can be split into $\xrightarrow{SR,*} \cdot \xrightarrow{I,M'_{inf},NSR}$ Note that a component with an infinite *SR*-reduction cannot have further *SR*-reductions in the reduction sequence.

Lemma 6.15. *A reduction $T \xrightarrow{I,M} T'$ can be split into its finite SR-components, and a reduction containing infinite SR-components and non-SR-components as follows: $T \xrightarrow{SR,*} T_1 \xrightarrow{I,M_1,NSR} T_2 \xrightarrow{I,M_2,infSR} T'$, where M_2 contains all positions that are in components with an infinite SR-reduction, and M_1 are the positions within the other process components.*

Proof. As already argued, the reductions in $T \xrightarrow{I,M} T'$ are parallel on the process components and can be split for the different components. There are the two cases of an infinite or finite *SR*-reduction for every component.

Lemma 6.16. *A reduction sequence $T \xrightarrow{I,M,infSR} T_1 \xrightarrow{SR,k} T_2$ can be commuted as $T \xrightarrow{SR,k} T'_1 \xrightarrow{I,M,infSR} T_2$.*

Lemma 6.17. *Consider two reductions* $\xrightarrow{I,M_1}$ *and* $\xrightarrow{I,M_2}$ *of type (betaTr), (caseTr) or (seqTr). For all tree processes* T, T_1, T_2*: if* $T \xrightarrow{I,M_1} T_1$*, and* $T \xrightarrow{I,M_2} T_2$*, and* $M_2 \subseteq M_1$*, then there is a set* M_3 *of positions, such that* $T_2 \xrightarrow{I,M_3} T_1$*.*



Proof. The argument is that the set M_3 is computed by labeling the positions in T using M_1, and then by performing the infinite development using the set of redexes M_2, where we assume that the M_1-labels are inherited. The set of positions of marked redexes in T_2 that remain and are not reduced by $T_1 \xrightarrow{I,M_2} T_2$ is exactly the set M_3.

Lemma 6.18. *Let* $T \xrightarrow{I,M,NSR} T_1 \xrightarrow{SR} T'$*. Then the reduction can be commuted to* $T \xrightarrow{SR} T_3 \xrightarrow{I,M'} T'$ *for some* M'*.*

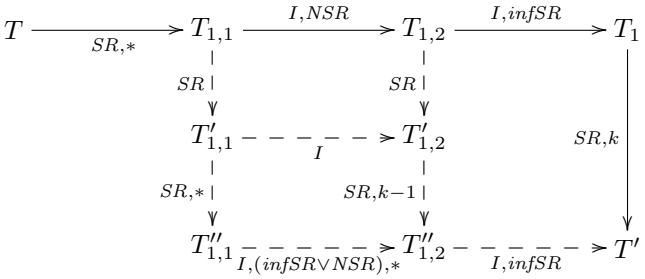
Proof. The *NSR*-reduction can only be a functional reduction, and since the *NSR*-reduction can only interact with the *SR*-reduction in deep positions or in other process components, it is easy to construct the reduction sequence $T_3 \xrightarrow{I,M'} T'$.

Lemma 6.19. *Let* $T \xrightarrow{I,M,infSR} T'$*, or* $T \xrightarrow{I,M,NSR} T'$*, where* T' *is successful. Then* T *is successful.*

Proof. This follows from the definitions of the reductions, since neither $\xrightarrow{I,M,infSR}$ nor $\xrightarrow{I,M,NSR}$ can generate a successful main thread.

Lemma 6.20. *Let* $T \xrightarrow{I} T_1 \xrightarrow{SR,k} T'$*. Then there is a reduction* $T \xrightarrow{SR,*} T_2 \xrightarrow{I,(infSR \vee NSR),*} T'$*.*

Proof. By induction on k:
The base case $k = 0$ is shown in Lemma 6.18.
Now let $k \geq 1$. The reduction $T \xrightarrow{I} T_1$ can be split according to Lemma 6.15 into $T \xrightarrow{SR,*} T_{1,1} \xrightarrow{I,M_1,NSR} T_{1,2} \xrightarrow{I,M_2,infSR} T_1$, where M_2 contains all positions that are in components with an infinite *SR*-reduction, and M_1 are the positions within the other process components.



Using Lemma 6.16, we see that $T_{1,2} \xrightarrow{SR,k} T''_{1,2} \xrightarrow{I,infSR} T'$ can be constructed. We split into the first and the other reductions and obtain: $T_{1,2} \xrightarrow{SR} T'_{1,2} \xrightarrow{SR,k-1} T''_{1,2} \xrightarrow{infSR} T'$. Now we commute the *NSR*-reduction with the first *SR*-reduction using Lemma 6.18. The reduction sequence is $T_{1,1} \xrightarrow{SR} T'_{1,1} \xrightarrow{I} T'_{1,2}$, and we can use induction on k and the reduction sequence $T'_{1,1} \xrightarrow{I} T'_{1,2} \xrightarrow{SR,k-1} T''_{1,2}$ to construct $T'_{1,1} \xrightarrow{SR,*} T''_{1,1} \xrightarrow{I,(infSR \vee NSR),*} T''_{1,2}$.

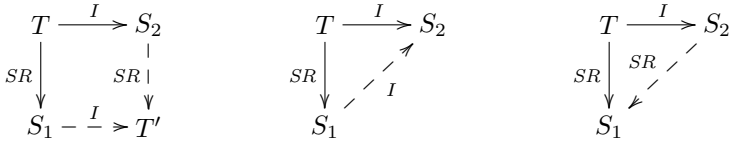
Proposition 6.21. *Let P be process such that $P{\downarrow}$. Then $IT(P){\downarrow}$.*

Proof. We assume that $P \xrightarrow{sr,*} P'$ where P' is successful. Using Lemma 6.11, we see that there is a finite sequence of reductions $IT(P) \xrightarrow{I,*} IT(P')$. Using induction on the number of the reduction $\xrightarrow{I}$, we show that there is an *SR*-reduction to a successful process. The induction step is to rearrange $T \xrightarrow{I} T_1 \xrightarrow{SR,k} T'$ where T' is successful. Lemma 6.20 shows that the reduction can be commuted such that we obtain the following reduction sequence $T \xrightarrow{SR,*} T'_1 \xrightarrow{I,(infSR \vee NSR),*} T'$. The properties of the reduction $\xrightarrow{I,(infSR \vee NSR),*}$ and that T' is successful imply that also T'_1 is successful by Lemma 6.19. By induction on the number of $\xrightarrow{I}$-reductions we obtain $IT(P){\downarrow}$.

6.2.2 Infinite Tree Convergence Implies Call-by-Need Convergence.

We now show that for every process P: if $IT(P)$ may-converges, then P may-converges, too.

Lemma 6.22. *Any overlapping between a $\xrightarrow{SR}$ and an $\xrightarrow{I,a}$-reduction, where is not (tmvar) and not (pmvar), can be closed as follows. The trivial case that both given reductions are identical is omitted. We have the following forking diagrams for infinite processes between an SR-reduction and an $\xrightarrow{I,a}$-reduction, where a is not (tmvar) and not (pmvar).*



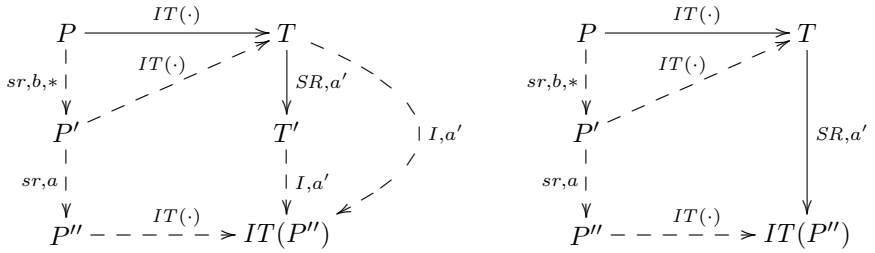
Proof. This follows by checking the overlaps of $\xrightarrow{I}$ with *SR*-reductions. Note that if the type of the $\xrightarrow{I}$ and $\xrightarrow{SR}$ reductions are different, then the first diagram applies.

Note that for `takeMVar` and `putMVar` the diagram cannot be closed if the redexes are in different processes and use the same Mvar.

Lemma 6.23. *Let T be an infinite process such that there is an SR-reduction sequence to a successful process of length n, and let S be an infinite process with $T \xrightarrow{I,a} S$, where a is not* `takeMVar` *and not* `putMVar`*. Then S has an SR-reduction sequence to a successful process of length $\leq n$.*

Proof. This follows from Lemma 6.22 by induction.

Lemma 6.24. *Let P be a process and let $T := IT(P) \xrightarrow{a'} T'$ be an SR-reduction. Then there is a process P', a reduction $P \xrightarrow{sr,*} P'$ using (mkbinds), (cp) and (cpcx)-reductions, a process P'' with $P' \xrightarrow{sr,a} P''$, where a is the process reduction corresponding to a', such that there is a reduction $T' \xrightarrow{I,a'} IT(P'')$. In the case of a monadic reduction a' we can even choose $T' = IT(P'')$.*



Proof. The processes P', P'' are constructed as follows: P' is the resulting process from a maximal sr-reduction of P consisting only of (cp), (cpcx) and (mkbinds)-reductions. Similarly as in the proof of Lemma 6.8 we can constructor a finite sr-reduction sequence $\xrightarrow{(cpcx)\vee(cp)\vee(mkbinds),sr}$ triggered by the redex of the a-reduction. Then obviously, $T := IT(P) = IT(P')$. The sr-(a)-redex in P' and its reduction uniquely correspond to $T \xrightarrow{SR,a'} T'$ and is used for the reduction $P' \xrightarrow{sr,a} P''$.

For functional rules, further arguments are required: Note that the (a)-redex in P' may correspond to infinitely many redexes in T. Lemma 6.11 shows that there is a reduction $T \xrightarrow{I,a'} IT(P'')$, and Lemma 6.17 shows that also $T' \xrightarrow{I,a'} IT(P'')$.

Proposition 6.25. *Let P be a process such that $IT(P)\downarrow$. Then $P\downarrow$.*

Proof. The precondition $IT(P)\downarrow$ implies that there is an *SR*-reduction sequence of $IT(P)$ to a successful process. The base case, where no *SR*-reductions are necessary is treated in Lemma 6.8. In the general case, let $T \xrightarrow{SR,a'} T'$ be an *SR*-reduction of a single redex. Lemma 6.24 shows that there are processes P', P'' with $P \xrightarrow{(cpcx)\vee(cp)\vee(mkbinds),sr,*} P' \xrightarrow{sr,a} P''$, and $T' = IT(P'')$ for a monadic reduction, and $T' \xrightarrow{I} IT(P'')$ for a functional reduction. For a functional reduction, Lemma 6.23 shows that the number of *SR*-reductions of $IT(P'')$ to a

successful process is strictly smaller than the number of SR-reductions of T to a successful process. For both kinds of reductions we can use induction on this length and obtain a sr-reduction of P to a successful process.

Propositions 6.21 and 6.25 imply the theorem

Theorem 6.26. *Let P be a process. Then $P{\downarrow}$ if and only if $IT(P){\downarrow}$.*

6.2.3 Equivalence w.r.t Should-Convergence. We now consider the should-convergence predicates for processes and infinite trees and show their coincidence. We mostly work with the negation of should-convergence, i.e. we show most of the results for may-divergence. The equivalence of convergence in Theorem 6.26 now implies the base case for inductions for may-divergence:

Corollary 6.27. *For a process P, we have $P{\Uparrow} \iff IT(P){\Uparrow}$*

Lemma 6.28. *Let T be an infinite process. If $T{\Uparrow}$ and $T \xrightarrow{a} T'$ for a reduction (caseTr), (seqTr), (betaTr), or a monadic SR-reduction, then $T'{\Uparrow}$.*

Proof. For the monadic computations the claim follows, since the monadic reductions are SR-reductions. If the reduction is one of the functional ones, then assume for contradiction that $T'{\downarrow}$. Lemma 6.20 shows that for a reduction $T' \xrightarrow{SR,*} T_1$ with T_1 successful, there is a reduction $T \xrightarrow{SR,*} T_2 \xrightarrow{I,(infSR \vee NSR),*} T_1$ and the same reasoning as in the proof of Proposition 6.21 shows that T_2 is successful, hence $T{\downarrow}$, which is a contradiction. Hence $T'{\Uparrow}$.

Lemma 6.29. *Let T be an infinite process. If $T \xrightarrow{I,M} T'$ and $T'{\Uparrow}$ for a reduction (caseTr), (seqTr), (betaTr), then $T{\Uparrow}$.*

Proof. Assume that $T \xrightarrow{I,M} T'$, and $T'{\Uparrow}$ and $T{\downarrow}$. The diagrams in Lemma 6.22 can be used for the induction on the length of an SR-reduction showing $T \xrightarrow{SR,*} T_1 \xrightarrow{I} T_2$ and $T' \xrightarrow{SR,*} T_2$, where T_1 is successful. Then also T_2 is successful, and we have a contradiction. Thus $T{\Uparrow}$.

Proposition 6.30. *Let P be a process with $P{\uparrow}$. Then also $IT(P){\uparrow}$.*

Proof. The reduction sequence $P \xrightarrow{sr,*} P'$ with $P'{\Uparrow}$ is translated into a sequence $IT(P) \xrightarrow{I,M,*} IT(P')$ where $IT(P'){\Uparrow}$ according to Corollary 6.27.

Using induction on the number of the reductions $\xrightarrow{I,*}$, we show that there is a SR-reduction to a must-divergent process. The induction step is to rearrange $T \xrightarrow{I} T_1 \xrightarrow{SR,k} T'$ where $T'{\Uparrow}$. Lemma 6.20 shows that it can be rearranged to $T \xrightarrow{SR,*} T_2 \xrightarrow{I,(infSR \vee NSR),*} T'$. Lemma 6.29 shows that $T_2{\Uparrow}$, since $\xrightarrow{I,(infSR \vee NSR),*}$ can only be (caseTr), (betaTr), and (seqTr)-reductions.

This concludes the induction on $\xrightarrow{I}$-reductions and we obtain $IT(P){\uparrow}$.

Lemma 6.31. *Let T be an infinite process such that there is a SR-reduction sequence of length n to a must-divergent process, and let S be a process with $T \xrightarrow{I,a} S$, where a is a (caseTr), (betaTr), or (seqTr)-reduction. Then $S \xrightarrow{SR,m} S'\Uparrow$ with $m \leq n$.*

Proof. The diagrams in Lemma 6.22 imply that there is a reduction sequence $S \xrightarrow{SR,m} S' \xrightarrow{I} S\Uparrow$ with $m \leq n$. Then Lemma 6.29 shows that $S'\Uparrow$.

Proposition 6.32. *Let P be a process such that $IT(P)\uparrow$. Then $P\uparrow$.*

Proof. The precondition $IT(P)\uparrow$ implies that there is an SR-reduction sequence of $T := IT(P)$ to a must-divergent process. The base case, where no SR-reductions are necessary is treated in Corollary 6.27.
In the general case, let $T \xrightarrow{SR,a'} T'$ be an SR-reduction.
Lemma 6.24 shows that there are processes P', P'' with $P \xrightarrow{(cpcx)\vee(cp)\vee(mkbinds),sr,*} P' \xrightarrow{sr,a} P''$, and if a' is a monadic reduction then $T' = IT(P'')$ and if a' is a functional reduction, then $T' \xrightarrow{I} IT(P'')$. For a functional reduction, Lemma 6.31 shows that the number of SR-reductions of $IT(P'')$ to a must-divergent process is strictly smaller than the number of SR-reductions of T to a must-divergent process. For both kinds of reductions we can use induction on this length and obtain a sr-reduction of P to a must-divergent process.

Proposition 6.30 and 6.32 imply the theorem

Theorem 6.33. *Let P be a process. Then $P\uparrow$ if and only if $IT(P)\uparrow$.*

Summarizing, we obtain the theorem:

Theorem 6.34. *Let P be a process. Then $P\downarrow$ if and only if $IT(P)\downarrow$ and $P\uparrow$ if and only if $IT(P)\uparrow$.*

6.3 Correctness of General Copy

A consequence of the former theorem is that we can use infinite trees and infinite tree convergences to prove contextual equivalences. Since the rules (cp) and $(cpcx)$ applied to processes do not modify the corresponding infinite trees we immediately have:

Proposition 6.35. *The reduction rules (cp) and (cpcx) are correct program transformations.*

Proof. Let $P_1 \xrightarrow{a} P_2$ where $a \in \{(cp), (cpcx)\}$. Then for every $PCtxt$-context $\mathbb{D}$ the equation $IT(\mathbb{D}[P_1]) = IT(\mathbb{D}[P_2])$ holds. Applying Theorem 6.34 yields $\mathbb{D}[P_1]\downarrow \iff \mathbb{D}[P_2]\downarrow$ and $\mathbb{D}[P_1]\Downarrow \iff \mathbb{D}[P_2]\Downarrow$. Thus we have $P_1 \sim_c P_2$.

We can generalize this result.

Let the general copy rule for processes be defined as

$$\text{(gcp)} \qquad \mathbb{C}[x] \mid x = e \to \mathbb{C}[e] \mid x = e$$

Theorem 6.36. *The general copy rule (gcp) is correct.*

Proof. Let $\mathbb{D}[P_1] \xrightarrow{\mathbb{D},gcp} \mathbb{D}[P_2]$. Since $IT(\mathbb{D}[P_1]) = IT(\mathbb{D}[P_2])$ and by Theorem 6.34, we obtain that $\mathbb{D}[P_1]{\downarrow} \iff \mathbb{D}[P_2]{\downarrow}$ and $\mathbb{D}[P_1]{\Downarrow} \iff \mathbb{D}[P_2]{\Downarrow}$. Hence we obtain $P_1 \sim P_2$.

6.4 Correctness of Call-By-Name Reduction for Processes

In this section we present a variant of the standard reduction which uses a call-by-name instead of the call-by-need strategy. We show that the reduction does not change the convergence predicates. The call-by-name reduction will then be helpful for proving correctness of the monad laws in the subsequent section.

The call-by-name standard reduction is a variant of the standard reduction $\xrightarrow{sr}$ where the rules (cp) and (cpcx) are replaced by (cpce) and by using replacing variants of (beta) and (case).

Definition 6.37 (Call-by-name Standard Reduction). *The rules of the call-by-name standard reduction $\xrightarrow{src}$ are defined in Fig. 5. We additionally assume that the rules are closed w.r.t. PCtxt and structural congruence, i.e. if $P \equiv \mathbb{D}[P']$, $Q \equiv \mathbb{D}[Q']$ and $P' \xrightarrow{src} Q'$ then also $P \xrightarrow{src} Q$.*

We also use the convergence predicates with their obvious definitions: The predicates ${\downarrow_{src}}$ and ${\Downarrow_{src}}$ denote for may- and should-convergence, and the predicates ${\Uparrow_{src}}$ and ${\uparrow_{src}}$ denote must- and may-divergence.

Now we show equivalence of call-by-name and call-by-need reduction in a series of lemmas. The technique is to show that call-by-name convergences coincide with tree convergences.

Lemma 6.38. *Let P be a process and let $T := IT(P) \xrightarrow{a'} T'$ be an SR-reduction. Then there is a process P', a reduction $P \xrightarrow{src,*} P'$ using (mkbinds) and (cpce)-reductions, a process P'' with $P' \xrightarrow{src,a} P''$, where a is the process reduction corresponding to a', such that there is a reduction $T' \xrightarrow{I,a'} IT(P'')$.*

Monadic Computations

(lunit) $y \Leftarrow \mathbb{M}[\texttt{return}\ e_1\ \texttt{>>=}\ e_2] \xrightarrow{src} y \Leftarrow \mathbb{M}[e_2\ e_1]$

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

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

(nmvar) $y \Leftarrow \mathbb{M}[\texttt{newMVar}\ e] \xrightarrow{src} \nu x.(y \Leftarrow \mathbb{M}[\texttt{return}\ x] \mid x\,\mathbf{m}\,e)$

(fork) $y \Leftarrow \mathbb{M}[\texttt{forkIO}\ e] \xrightarrow{src} \nu z.(y \Leftarrow \mathbb{M}[\texttt{return}\ z] \mid z \Leftarrow e)$
where z is fresh and the created thread is not a main thread

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

Functional Evaluation

(cpce) $y \Leftarrow \mathbb{M}[\mathbb{F}[x]] \mid x = e \xrightarrow{src} y \Leftarrow \mathbb{M}[\mathbb{F}[e]] \mid x = e$

(mkbinds) $y \Leftarrow \mathbb{M}[\mathbb{F}[\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e]]$
$\xrightarrow{src} \nu x_1, \ldots, x_n.(y \Leftarrow \mathbb{M}[\mathbb{F}[e]] \mid x_1 = e_1 \mid \ldots \mid x_n = e_n)$

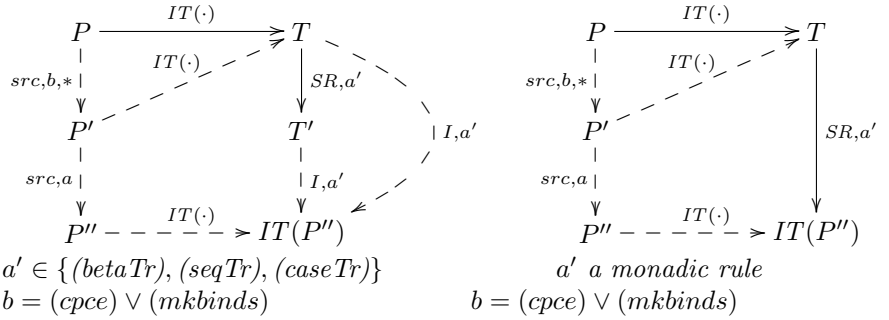
(nbeta) $y \Leftarrow \mathbb{M}[\mathbb{F}[((\lambda x.e_1)\ e_2)]] \xrightarrow{src} y \Leftarrow \mathbb{M}[\mathbb{F}[e_1[e_2/x]]]$

(ncase) $y \Leftarrow \mathbb{M}[\mathbb{F}[\texttt{case}_T\ (c\ e_1\ \ldots\ e_n)\ \texttt{of}\ \ldots((c\ y_1\ \ldots\ y_n) \to e)\ldots]]$
$\xrightarrow{src} y \Leftarrow \mathbb{M}[\mathbb{F}[e[e_1/y_1, \ldots, e_n/y_n]]]$

(seq) $y \Leftarrow \mathbb{M}[\mathbb{F}[(\texttt{seq}\ v\ e)]] \xrightarrow{src} y \Leftarrow \mathbb{M}[\mathbb{F}[e]]$ if v is a functional value

Fig. 5. Call-by-name reduction rules

In the case of a monadic reduction a', we can even choose $T' = IT(P'')$.



Proof. The processes P', P'' are constructed as follows: P' is the resulting process from an src-reduction of P consisting only of (cpce) and (mkbinds)-reductions. Similarly as in the proof of Lemma 6.8 we can construct this finite src-reduction sequence $\xrightarrow{(cpce)\vee(mkbinds),scr,*}$ triggered by the redex of the a'-reduction. Then obviously, $T = IT(P) = IT(P')$ since (cpce) and (mkbinds) do not change the infinite tree. The src-(a)-redex in P' and its reduction uniquely correspond to $T \xrightarrow{SR,a'} T'$ and is used for the reduction $P' \xrightarrow{src,a} P''$. For a monadic rule, we have $T' = IT(P'')$.

For functional rules, further arguments are required: Note that the (a)-redex in P' may correspond to infinitely many redexes in T. Lemma 6.11 shows that

there is a reduction $T \xrightarrow{I,a'} IT(P'')$, and Lemma 6.17 shows that also $T' \xrightarrow{I,a'} IT(P'')$.

Lemma 6.39. *Let P be a process. If P is successful then $IT(P)$ is successful. If $IT(P)$ is successful, then $P{\downarrow_{src}}$.*

Proof. If $IT(P)$ is successful, then $P \xrightarrow{(cpce)\vee(mkbinds),src,*} P'$, where P' is successful, hence $P{\downarrow}$.

Lemma 6.40. *For every process P: $P{\downarrow}$ iff $P{\downarrow_{src}}$.*

Proof. The claim is shown by transferring it into the infinite processes. We show that for all P: $P{\downarrow_{src}}$ iff $IT(P){\downarrow}$.

If $P{\downarrow_{src}}$, then there is a reduction $IT(P) \xrightarrow{I,*} T'$ where T' is successful. The proof of Proposition 6.21 shows that $IT(P){\downarrow}$.

To show the other direction, let $IT(P){\downarrow}$. The precondition $IT(P){\downarrow}$ implies that there is an *SR*-reduction sequence of $IT(P)$ to a successful process. The base case, where no *SR*-reductions are necessary is treated in Lemma 6.39. In the general case, let $T \xrightarrow{SR,a'} T'$ be an *SR*-reduction. Lemma 6.38 shows that there are processes P', P'' with $P \xrightarrow{(cpce)\vee(mkbinds),src,*} P' \xrightarrow{src,a} P''$, and $T' = IT(P'')$ for a monadic reduction, and $T' \xrightarrow{I} IT(P'')$ for a functional reduction. For a functional reduction, Lemma 6.23 shows that the number of *SR*-reductions of $IT(P'')$ to a successful process is strictly smaller than the number of *SR*-reductions of T to a successful process. For both kinds of reductions we can use induction on this length and obtain a src-reduction of P to a successful process.

Lemma 6.41. *Let P be a process. Then $P{\Uparrow_{src}}$ iff $IT(P){\Uparrow}$.*

Proof. This follows from 6.39 by logical operations.

Lemma 6.42. *For every process P: $P{\uparrow}$ iff $P{\uparrow_{src}}$.*

Proof. Again the claim is shown by transferring it into the infinite processes. We show that for all P: $P{\uparrow_{src}}$ iff $IT(P){\uparrow}$.

If $P{\uparrow_{src}}$, then there is a reduction $IT(P) \xrightarrow{I,*} T'$ where $T'{\Uparrow}$. The proof of Proposition 6.30 shows that $IT(P){\uparrow}$.

To show the other direction, let $IT(P){\uparrow}$. The precondition $IT(P){\uparrow}$ implies that there is an *SR*-reduction sequence of $IT(P)$ to a must-divergent process. The base case, where no *SR*-reductions are necessary is treated in Lemma 6.41. In the general case, let $T \xrightarrow{SR,a'} T'$ be an *SR*-reduction. Lemma 6.38 shows that there are processes P', P'' with $P \xrightarrow{(cpce)\vee(mkbinds),src,*} P' \xrightarrow{src,a} P''$, and $T' = IT(P'')$ for a monadic reduction, and $T' \xrightarrow{I} IT(P'')$ for a functional reduction. For a functional reduction, Lemma 6.23 shows that the number of *SR*-reductions of $IT(P'')$ to a must-divergent process is strictly smaller than the number of *SR*-reductions of T to a must-divergent process. For both kinds of reductions we can use induction on this length and obtain a src-reduction of P to a must-divergent process.

Theorem 6.43. *The call-by-name reduction* $\xrightarrow{src}$ *is equivalent to the reduction* $\xrightarrow{sr}$*. I.e., for every process* P*:* $P{\downarrow}$ *iff* $P{\downarrow_{src}}$ *and* $P{\Downarrow}$ *iff* $P{\Downarrow_{src}}$*.*

Proof. This follows from Lemmas 6.42 and 6.40 and Theorem 6.34.

The context lemma 4.27 also holds for the call-by-name reduction.

Lemma 6.44 (Context Lemma for Call-By-Name and Expressions). *Let* e_1, e_2 *be expressions of type* τ *such that for all* $\mathbb{D} \in PCtxt$ *and* $\mathbb{L}[\cdot^\tau] \in LCtxt$*:* $\mathbb{D}[\mathbb{L}[e_1]]{\downarrow_{src}} \implies \mathbb{D}[\mathbb{L}[e_2]]{\downarrow_{src}}$ *and* $\mathbb{D}[\mathbb{L}[e_1]]{\Downarrow_{src}} \implies \mathbb{D}[\mathbb{L}[e_2]]{\Downarrow_{src}}$*. Then* $e_1 \leq_{c,\tau} e_2$*.*

Proof. This follows from the context lemma 4.27 and the equivalence of call-by-need and call-by-name-reductions in Theorem 6.43.

7 Monad Laws

We show in this subsection that the three monad laws are correct for $\sim_c$, i.e. the laws are:

$$\begin{array}{ll}
\text{(M1) } \texttt{return}\ e_1\ \texttt{>>=}\ e_2 & = e_2\ e_1 \\
\text{(M2) } e_1\ \texttt{>>=}\ \lambda x.\texttt{return}\ x & = e_1 \\
\text{(M3) } e_1\ \texttt{>>=}\ (\lambda x.(e_2\ x\ \texttt{>>=}\ e_3)) & = (e_1\ \texttt{>>=}\ e_2)\ \texttt{>>=}\ e_3
\end{array}$$

Note, that the monad law (M1) is analogous to our reduction rule (lunit), but defined on expressions.

Remark 7.1. The monad laws would be incorrect if `seq` can be used without restrictions: Assume that the first argument of `seq` is not type restricted. Also we adopt the natural assumption that the monadic operators are treated like constructor in `seq`, i.e., (`seq` $(c\ \ldots)\ s$) reduces to s for the monadic operators c. This behavior can also be observed in the GHC implementation of Haskell. Let `undefined` be a diverging closed expression, e.g. `letrec` $x = x$ `in` x.

The law (M1) does not hold under unrestricted `seq`:
(`seq` (`return True >>= undefined`) `True`) terminates, but (`undefined True`) does not terminate.

The law (M2) does not hold under unrestricted `seq`:
(`seq` (`undefined >>=` $\lambda x.$`return` x) `True`) is permitted under unrestricted `seq`. Since the operator `>>=` is treated like a constructor, it will result in `True`. On the other hand, the monad law implies that this expression is equivalent to `seq` (`undefined`) `True`, which does not terminate.

However, due to our restriction that the first argument of `seq` cannot be of type (`IO` a), the monad laws are valid in our calculus, as we will show.

7.1 Restricted Contexts and Commutation

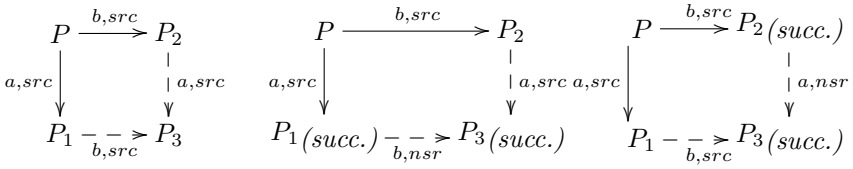
We show that the three monad laws are correct, where we use the call-by-name reduction strategy $\xrightarrow{src}$ for usual processes in the proofs and also for computing the diagrams.

We define a further class of contexts: $\mathbb{A}, \mathbb{A}_i \in \mathit{ACtxts} ::= x \Leftarrow \mathbb{M} \mid x = \mathbb{M}$, where $\mathbb{M} \in \mathit{MCtxt}$. Let (M1A), (M2A), (M3A) be the reductions that permit to perform (M1), (M2), (M3) in arbitrary $\mathbb{D}[\mathbb{A}[\cdot]]$-contexts. We write $\xrightarrow{M1A,nsr}$ iff the reduction is an $(M1A)$-reduction, but not an $(lunit, src)$-reduction. We also use $\xrightarrow{M1A,src}$ for an $(lunit, src)$-reduction. We also use the notation $\xrightarrow{a,nsr}$ for the reduction rules of the call-by-name reduction, meaning that the reduction rule a is applied, but not as an src-reduction, since either the context is not a $\mathbb{M}$- or $\mathbb{M}[\mathbb{F}]$-context, or the process where the rule is applied is already successful.

Lemma 7.2. *If an expression $t :: \mathtt{IO}\ \tau$ is in a $\mathbb{D}[\mathbb{L}[\cdot]]$-context, then the context is a $\mathbb{D}[\mathbb{A}[\cdot]]$-context.*

Proof. The type system in Fig. 2 shows that all other contexts are ruled out by the type restrictions.

Lemma 7.3. *For all P: if $P \xrightarrow{a,src} P_1$ and $P \xrightarrow{b,src} P_2$, where the reductions are at different redexes, and where a, b are not the same kind of src-reduction (pmvar) or (tmvar) on the same MVar. Then there are three cases: If neither P_1 nor P_2 are successful, then there is some P_3, such that $P_1 \xrightarrow{b,src} P_3$ and $P_2 \xrightarrow{a,src} P_3$. If P_1 is successful, then there is some P_3 with $P_1 \xrightarrow{b,nsr} P_3$ and $P_2 \xrightarrow{a,src} P_3$. If P_2 is successful, then there is some P_3 with $P_1 \xrightarrow{b,src} P_3$ and $P_2 \xrightarrow{a,nsr} P_3$.*



Proof. The call-by-name standard reductions are non-overlapping, since they make only changes in one thread with the exception of the Mvar-modifying reductions.

Lemma 7.4. *Let $P \xrightarrow{b,src} P'$ where $b \notin \{(pmvar), (tmvar)\}$, then the following holds:*

1. *If $P \xrightarrow{src,k} P_0$ where P_0 is successful, then there is some successful P_0' and $m \leq k$ with $P' \xrightarrow{src,m} P_0'$.*
2. *If $P' \xrightarrow{src,k} P_0'$ where P_0' is successful, then there is some successful P_0 and $m \leq k+1$ with $P \xrightarrow{src,m} P_0$.*
3. *If $P \xrightarrow{src,k} P_0$ where $P_0 \Uparrow$, then there is some must-divergent P_0' and $m \leq k$ with $P' \xrightarrow{src,m} P_0'$.*

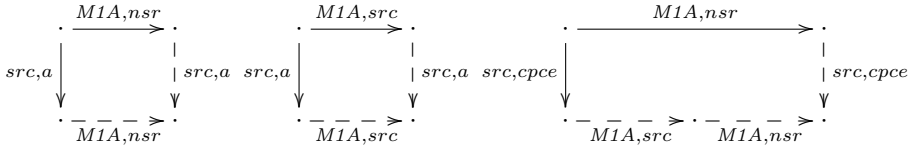
4. *If* $P' \xrightarrow{src,k} P'_0$ *where* $P'_0\Uparrow$ *, then there is some must-divergent* P_0 *and* $m \leq k+1$ *with* $P \xrightarrow{src,m} P_0$.

Proof. This follows by induction on k, and from the fact that standard reductions commute according to Lemma 7.3. In the induction proofs of the last two items, the results of the first two items are used.

7.2 Correctness of M1.

Now we show that the monad law (M1) is correct.

Lemma 7.5. *The overlappings for (M1A) with src-reductions are as follows, where we assume that the (M1A) and the src-reduction is different.*



Proof. Note that occurrences of the type `IO` τ are severely restricted. The only nontrivial overlap is with the (*cpce*)-rule which generates the second and last diagram. The prototypical overlap is:

$$\begin{array}{ccccc} \mathbb{C}[x] & & \xrightarrow{\quad M1A,nsr \quad} & & \mathbb{C}[x] \mid x = (e_2\ e_1) \\ \mid x = \texttt{return}\ e_1\ \texttt{>>=}\ e_2 & & & & \Big\downarrow {\scriptstyle src,cpce} \\ \Big\downarrow {\scriptstyle src,cpce} & & & & \\ \mathbb{C}[\texttt{return}\ a\ \texttt{>>=}\ e_2] & & \mathbb{C}[(e_2\ e_1)] & & \mathbb{C}[(e_2\ e_1)] \\ \mid x = \texttt{return}\ e_1\ \texttt{>>=}\ e_2 & \overset{M1A,src}{\dashrightarrow} & \mid x = \texttt{return}\ e_1\ \texttt{>>=}\ e_2 & \overset{M1A,nsr}{\dashrightarrow} & \mid x = (e_2\ e_1) \end{array}$$

Here the first bottom reduction must be an src-reduction, since the context $\mathbb{C}$ must be a $\mathbb{D}[\mathbb{F}]$-context.

Lemma 7.6. *Let* $P \xrightarrow{M1A} P'$ *with* $P\downarrow$. *Then also* $P'\downarrow$.

Proof. Let $P \xrightarrow{src,k} P_0$ be a reduction to a successful process P_0. We use the diagrams in Lemma 7.5 to show that $P' \xrightarrow{src,m} P'_0$ with $m \leq k$. Scanning the diagrams shows that the induction step is proved. The base case is that $P \xrightarrow{\text{M1A}} P'$ where P' is successful implies that P is successful.

Lemma 7.7. *Let* $P \xrightarrow{M1A} P'$ *with* $P'\downarrow$. *Then also* $P\downarrow$.

Proof. For the proof we interpret the diagrams "commuting diagrams", i.e. the given sequence is $\xrightarrow{M1A,nsr}$. $\xrightarrow{src,a}$. Note that we do not consider given $(M1a, src)$ reductions for this part. The diagrams also hold as commuting diagram which can be justified by the determinism within the threads. Now by

induction on the length of a successful ending reduction sequence for P' one can show that also $P{\downarrow}$. The base case obviously holds, the induction step follows by applying the induction hypothesis first. Then either a commuting diagram is applied or the $(M1A)$-reduction is also an src-reduction and the claim also holds.

Lemma 7.8. *Let $P \xrightarrow{M1A} P'$. Then $P'{\uparrow}$ iff $P{\uparrow}$.*

Proof. The base case, i.e. the equivalence $P'{\Uparrow}$ iff $P{\Uparrow}$ follows from Lemmas 7.6 and 7.7.
We have to prove two parts.
Let $P \xrightarrow{\text{M1A}} P'$ and $P \xrightarrow{*,src} P_1$ with $P_1{\Uparrow}$. The diagrams in Lemma 7.5 show that there is a process P_2 with $P_1 \xrightarrow{\text{M1A},*} P_2$, and $P' \xrightarrow{*,src} P_2$. Lemma 7.7 shows that $P_2{\Uparrow}$, hence this part is proved.
Let $P \xrightarrow{\text{M1A}} P'$ and $P' \xrightarrow{*,src} P_2$ with $P_2{\Uparrow}$. The diagrams are now interpreted as commuting diagrams (omitting the second diagram) and an induction on the length of the sequence $P' \xrightarrow{*,src} P_2$ using the diagrams and the induction hypothesis shows that $P{\uparrow}$.

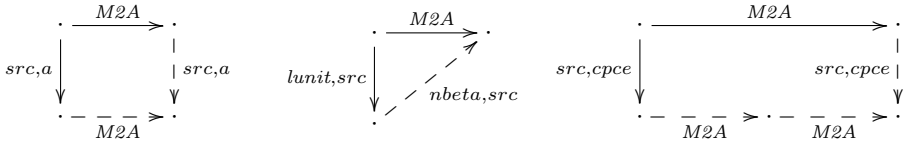
Together with the context lemma 6.44 and Lemma 7.2 we have proved:

Proposition 7.9. *The monad law (M1) is correct.*

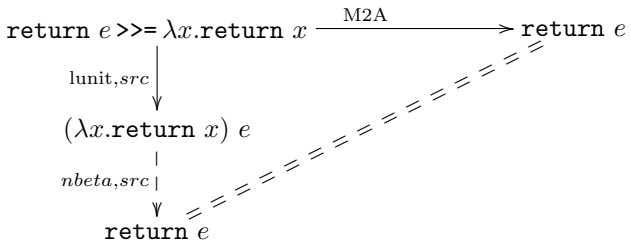
7.3 Correctness of M2

Let (M2A) be the reduction that permits to perform (M2) also in arbitrary $\mathbb{D}[\mathbb{A}[\cdot]]$-contexts.

Lemma 7.10. *The overlappings for (M2A) with src-reductions are:*



Proof. One non-trivial overlap is with the $(cpce, src)$-rule similar to the diagram in 7.5, and the other nontrivial overlap is with the $(lunit, src)$-rule which generates the second diagram.



Lemma 7.11. *Let* $P \xrightarrow{M2A} P'$. *Then* $P{\downarrow} \iff P'{\downarrow}$, *and* $P{\uparrow} \iff P'{\uparrow}$,

Proof. The diagrams in Lemma 7.10 are now used for a proof by induction:

1. $P{\downarrow} \implies P'{\downarrow}$: We show a stronger claim: If $P \xrightarrow{src,k} P_0$ where P_0 is successful, then there exists a successful process P'_0 with $P' \xrightarrow{src,m} P'_0$ such that $m \leq k$. The proof is by induction on k. For the base case it obviously holds that if P is successful, then P'_0 is successful. For the induction step let $P \xrightarrow{src} P_1 \xrightarrow{src,k-1} P_0$. We apply a diagram to $P_1 \xleftarrow{src} P \xrightarrow{M2A} P'$ and then apply the induction hypothesis (once for the first diagram, twice for the third diagram). The second diagram is covered by Lemma 7.3.
2. $P{\downarrow} \impliedby P'{\downarrow}$: Here we use the diagrams in Lemma 7.10 as commuting diagrams. The claim is: if $P' \xrightarrow{src} P'_0$ where P'_0 is successful and there are n (cpce,src)-reductions, then $P' \xrightarrow{src,*} P_0$ where P_0 is successful using at most n (cpce,src)-reductions. The induction is on the reduction $P' \xrightarrow{src} P'_0$, where the measure is the number of (cpce,src)-reductions, then the total number of reductions. For the third diagram the induction is applicable, since the number of (cpce,src)-reductions is strictly decreased, and thus we can apply the hypothesis twice. If the first diagram is applied, then the number of reductions is strictly decreased, and for the second, the diagram can be immediately applied.

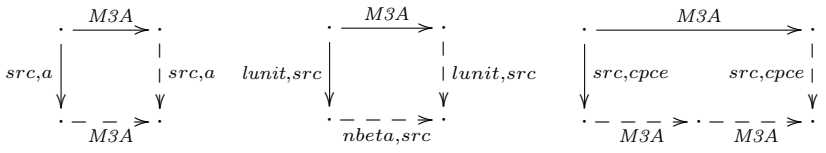
   The base case is that P' is successful, and that $P \xrightarrow{M2A} P'$. Then $P \equiv \mathbb{D}[x \xleftarrow{\mathsf{main}} \texttt{return}\ e_1 \texttt{>>=} \lambda y.\texttt{return}\ y]$ and $P' \equiv \mathbb{D}[x \xleftarrow{\mathsf{main}} \texttt{return}\ e_1]$. Then the second diagram shows that $P \xrightarrow{(lunit)\vee(nbeta),src,*} P'$.
3. $P{\uparrow} \implies P'{\uparrow}$: We derive $P{\Uparrow} \iff P'{\Uparrow}$ using the first two items. If $P{\uparrow}$, then again the diagrams and a simple induction show the claim using the base case.
4. $P{\uparrow} \impliedby P'{\uparrow}$: Using a similar reasoning as in item (2), i.e., the corresponding induction claim and the same induction measure and using $P{\Uparrow} \iff P'{\Uparrow}$, an induction shows the claim.

Together with the context lemma 6.44 and Lemma 7.2 we have proved:

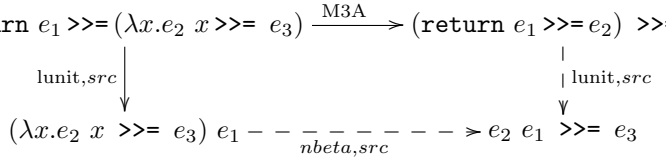
Proposition 7.12. *The monad law (M2) is correct w.r.t.* $\sim_c$.

7.4 (M3) is correct

Lemma 7.13. *The overlappings for (M3A) with src-reduction are:*



Proof. The nontrivial overlaps are with the (*lunit*)-rule and the (*cpce*)-rule.



Lemma 7.14. *Let $P \xrightarrow{M3A} P'$. Then $P\downarrow$ iff $P'\downarrow$ and $P\Downarrow$ iff $P'\Downarrow$.*

Proof. The diagrams in Lemma 7.13 can be used for a proof by induction, very similar to the proof for M2A in Lemma 7.11. The following differences have to be obeyed: M3A cannot turn a non-successful process into a successful one, unlike M2A, and in the second diagram an (lunit,src) is there instead of an equality, which is only a minor difference. The proof is almost the same.

Together with Lemmas 6.44 and 7.2 we have proved:

Proposition 7.15. *The monad law (M3) is correct w.r.t. $\sim_c$.*

Propositions 7.9, 7.12, and 7.15 show:

Theorem 7.16. *The monad laws (M1), (M2), and (M3) are correct w.r.t. $\sim_c$.*

8 Conclusion and Further Work

We presented the calculus CHF as a model for Concurrent Haskell extended by futures. We have shown the correctness of a lot of program transformations. In particular we have shown that call-by-name evaluation is correct, which opens a wide range of further program optimizations. We use a monomorphic type system, but we are convinced that our results can be transferred to polymorphic typing. We have shown that the monad laws are correct in CHF, but we needed to restrict the first argument of the `seq`-operator to function types and constructor types. This result also applies to usual (sequential) Haskell, since the monad laws for the `IO`-monad can be falsified using `seq`if the first argument may be an action of IO-type.

Ongoing work is to prove that CHF is "referential transparent", that is we try to show that pure functions that are equivalent in a pure call-by-need calculus (without IO and threads) are also equivalent in CHF.

Acknowledgments

We thank Conrad Rau for reading a version of this paper.

References

Abr90. Samson Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

Ali11. Alice ML. Homepage, 2011. `http://www.ps.uni-saarland.de/alice/`.

AS98. Zena M. Ariola and Amr Sabry. Correctness of monadic state: An imperative call-by-need calculus. In *POPL 98*, pages 62–74, 1998.

Bar84. H. P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam, New York, 1984.

BFKT00. Clement A. Baker-Finch, David J. King, and Philip W. Trinder. An operational semantics for parallel lazy evaluation. In *International Conference on Functional Programming, ICFP*, pages 162–173. ACM Press, 2000.

BH77. Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, New York, NY, USA, 1977. ACM.

CHS05. Arnaud Carayol, Daniel Hirschkoff, and Davide Sangiorgi. On the representation of McCarthy's amb in the pi-calculus. *Theoret. Comput. Sci.*, 330(3):439–473, 2005.

DH84. R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoret. Comput. Sci.*, 34:83–133, 1984.

FF99. C. Flanagan and Mattias Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9:1–31, January 1999.

Gla11. Glasgow parallel Haskell. Homepage, 2011. `http://www.macs.hw.ac.uk/~dsg/gph/`.

Hal85. Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, October 1985.

HMPJH05. Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.

KKSdV97. Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. Infinitary lambda calculus. *Theoret. Comput. Sci.*, 175(1):93–125, 1997.

LOMPM05. Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña-Marí. Parallel functional programming in Eden. *J. Funct. Program.*, 15(3):431–475, 2005.

McC63. John McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

Mil99. Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge university press, 1999.

MML+10. Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. Seq no more: better strategies for parallel haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 91–102, New York, NY, USA, 2010. ACM.

Mor68. J.H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.

Moz11. Mozart. Homepage, 2011. `http://www.mozart-oz.org/`.

NC95. V. Natarajan and Rance Cleaveland. Divergence and fair testing. In *ICALP 1995*, volume 944 of *Lecture Notes in Comput. Sci.*, pages 648–659. Springer, 1995.

NSS06. Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoret. Comput. Sci.*, 364(3):338–356, November 2006.

NSSSS07. Joachim Niehren, David Sabel, Manfred Schmidt-Schauß, and Jan Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *Electron. Notes Theor. Comput. Sci.*, 173:313–337, 2007.

Pey01. Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Ralf Steinbruggen Tony Hoare, Manfred Broy, editor, *Engineering theories of software construction*, pages 47–96. IOS-Press, 2001. Presented at the 2000 Marktoberdorf Summer School.

Pey03. Simon Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003. `www.haskell.org`.

PGF96. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. 23th Principles of Programming Languages*, 1996.

Plo75. Gordon D. Plotkin. Call-by-name, call-by-value, and the lambda-calculus. *Theoret. Comput. Sci.*, 1:125–159, 1975.

PS09. Simon Peyton Jones and Satnam Singh. A tutorial on parallel and concurrent programming in haskell. In *Proceedings of the 6th international conference on Advanced functional programming*, AFP'08, pages 267–305, Berlin, Heidelberg, 2009. Springer-Verlag.

PW93. Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages, Charleston, South Carolina,*, pages 71–84. ACM, 1993.

RV07. Arend Rensink and Walter Vogler. Fair testing. *Inform. and Comput.*, 205(2):125–198, 2007.

SS07. Manfred Schmidt-Schauß. Correctness of copy in calculi with letrec. In *Term Rewriting and Applications (RTA-18)*, volume 4533 of *LNCS*, pages 329–343. Springer, 2007.

SSS08. David Sabel and Manfred Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 18(03):501–553, 2008.

SSS10. Manfred Schmidt-Schauß and David Sabel. Closures of may-, should- and must-convergences for contextual equivalence. *Information Processing Letters*, 110(6):232 – 235, 2010.

SSSH09. David Sabel, Manfred Schmidt-Schauß, and Frederik Harwath. Reasoning about contextual equivalence: From untyped to polymorphically typed calculi. In Stefan Fischer, Erik Maehle, and Rüdiger Reischuk, editors, *INFORMATIK 2009, Im Focus das Leben, Beiträge der 39. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 28.9 - 2.10.2009 in Lübeck*, volume 154 of *GI Edition - Lecture Notes in Informatics*, pages 369; 2931–45, October 2009. (4. Arbeitstagung Programmiersprachen (ATPS)).

SSSM10. Manfred Schmidt-Schauß, David Sabel, and Elena Machkasova. Simulation in the call-by-need lambda-calculus with letrec. In Christopher Lynch, editor, *Proc. of 21st RTA 2010*, volume 6 of *LIPIcs*, pages 295–310. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010.

SSSSN09. Jan Schwinghammer, David Sabel, Manfred Schmidt-Schauß, and Joachim Niehren. Correctly translating concurrency primitives. In *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 27–38, New York, NY, USA, August 2009. ACM.

SW01. D. Sangiorgi and D. Walker. *The π-calculus: a theory of mobile processes.* Cambridge university press, 2001.

THLP98. Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

Wad95. Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.