On Conservativity of Concurrent Haskell

David Sabel and Manfred Schmidt-Schauss

Goethe-University Frankfurt am Main, Germany

Technical Report Frank-47

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

October 17, 2011

Abstract. The calculus CHF models Concurrent Haskell extended by concurrent, implicit futures. It is a process calculus with concurrent threads, monadic concurrent evaluation, and includes a pure functional lambda-calculus which comprises data constructors, case-expressions, letrec-expressions, and Haskell's seq. Futures can be implemented in Concurrent Haskell using the primitive `unsafeInterleaveIO`, which is available in most implementations of Haskell. Our main result is conservativity of CHF, that is, all equivalences of pure functional expressions are also valid in CHF. This implies that compiler optimizations and transformations from pure Haskell remain valid in Concurrent Haskell even if it is extended by futures. We also show that this is no longer valid if Concurrent Haskell is extended by the arbitrary use of `unsafeInterleaveIO`.

1 Introduction

Pure nonstrict functional programming is semantically well understood, permits mathematical reasoning and is referentially transparent (see [29]). A witness is the core language of the functional part of Haskell [16] consisting only of supercombinator definitions, abstractions, applications, data constructors and case-expressions. However, useful programming languages require much more expressive power for controlling interaction with the operating system, the user, the file system and further computing devices. Haskell's expressiveness currently employs monadic programming [30, 20] as an interface between the imperative world and pure nonstrict functional programming. Sometimes the sequentialization of IO-operations enforced by Haskell's IO-monad is too strong a requirement to allow declarative programming. Implementations of Haskell often provide primitives which break the sequentialization to enable *lazy IO* [20, 17]. One such primitive is `unsafePerformIO :: IO a` $\rightarrow$ `a` which switches off any restrictions enforced by the IO-monad. Another one is `unsafeInterleaveIO :: IO a → IO a` which delays a monadic action inside the IO-monad: Given the program `do` $\{x_1 \leftarrow act_1; x_2 \leftarrow act_2; \ldots\}$ Haskell's IO-monad ensures that the actions act_1, act_2, ... are strictly executed in sequence where the results are bound to the variables $x_1, x_2, \ldots$. Wrapping `unsafeInterleaveIO` around action act_i breaks the strict sequencing, i.e. action act_i is performed at the time the value of x_i is *needed* and thus not necessarily before act_{i+1}.

Another extension is *Concurrent Haskell* [19,17,18]. It extends Haskell by the primitive `forkIO` which takes a monadic computation (of type `IO ()`) and immediately spawns a new thread to concurrently perform the computation. As synchronization primitives Concurrent Haskell provides synchronizing variables, called `MVar`s. An `MVar` is either empty or filled. The operation `newEmptyMVar` creates an empty `MVar`, the operation `takeMVar` reads the value of a filled `MVar` and empties it. Similarly, `putMVar` v e fills the empty `MVar` v with content e. `takeMVar` blocks on an empty MVar and `putMVar` blocks on a filled MVar.

For all these extensions of Haskell it is either obvious that they are unsafe (e.g. `unsafePerformIO`) or the situation is not well understood. For instance, Kiselyov [9] provides an example showing that the extension of pure Haskell by `unsafeInterleaveIO` is non-conservative, since side effects can be observed in the pure functional world. He exhibits two pure functions f, g that are semantically equal under pure functional semantics, but can be distinguished if they get their input through lazy file reading (implemented using `unsafeInterleaveIO`). This is awkward from a practical point of view, since it appears to indicate that soundness of a compiler for pure Haskell does not necessarily transfer to extensions, in particular certain optimizations and transformations performed by a Haskell-compiler on pure functional expressions may be wrong in extensions.

One possible way out of this dilemma is to use a precise semantics that models nondeterminism, sharing and laziness (see for example [23]) which could be extended to model impure and non-deterministic computations correctly, and then adapt the compiler accordingly.

We follow a different approach for laying the foundation of correct reasoning that exploits the separation between pure functional and impure computations by monadic programming. In [24] we introduced the process calculus *CHF*, a pure nonstrict functional language. *CHF* can be seen as a core language of Concurrent Haskell extended by implicit concurrent futures: Futures are variables whose value is initially not known, but becomes available in the future when the corresponding (concurrent) computation is finished (see e.g. [2,5]). *Implicit* futures do not require explicit forces when their value is demanded, and thus they permit a declarative programming style using implicit synchronization by data dependency. Implicit futures can be implemented in Concurrent Haskell using the extension by the `unsafeInterleaveIO`-primitive:

```
future :: IO a → IO a
future act = do ack ← newEmptyMVar
                thread ← forkIO (act >>= putMVar ack)
                unsafeInterleaveIO (takeMVar ack)
```

First an empty `MVar` is created, which is used to store the result of the concurrent computation. This computation is performed in a new concurrent thread spawned by using `forkIO`. The last part consists of taking the result of the MVar, which is delayed using `unsafeInterleaveIO`.

In *CHF* the above `future`-operation is built-in as a primitive. Unlike the π-calculus [12, 25] (which is a message passing model), the calculus *CHF* comprises shared memory modelled by MVars, threads (i.e. futures) and heap bindings. On the expression level *CHF* provides an extended lambda-calculus where the extensions are closely related to Haskell's core language: Expressions comprise data constructors, `case`-expressions, `letrec` to express recursive bindings, Haskell's `seq`-operator for sequential evaluation, and monadic operators for accessing MVars, creating futures, and the bind-operator `>>=` for monadic sequencing. *CHF* is equipped with a monomorphic type system allowing recursive types. In [24] two (semantically equivalent) small-step reduction strategies are introduced as operational semantics for *CHF*: A call-by-need strategy which avoids duplication by sharing and a call-by-name strategy which copies arbitrary subexpressions. The operational semantics of *CHF* is related to the operational semantics for Concurrent Haskell introduced in [11, 17] where also exceptions are considered. *CHF* also borrows some ideas from the impure call-by-value lambda calculus with futures [14, 15].

In [24] there are strong results about *CHF*, e.g. the monad laws have been proved to be correct (where the type of `seq` was restricted to functional types), but we had to leave open the important question whether the extension of Haskell by concurrency and futures is a *safe* extension. In this paper we address this question and obtain a positive result: *CHF* is a *conservative extension* of its pure sublanguage, i.e. the equality (following Abramsky [1]) of pure functional expressions transfers into the full calculus, where the semantics is defined as a contextual equality for a conjunction of may- and should-convergence. This result enables equational reasoning, pure functional transformations and optimizations also in the full concurrent calculus, *CHF*. Haskell's type system is polymorphic with type classes whereas *CHF* has a monomorphic type system. Nevertheless we believe that our main result can be transferred to the polymorphic case. Our results also imply that Kiselyov's [9] counterexample is not possible for *CHF*.

We also analyze the boundaries of our conservativity result and show that if so-called *lazy futures* (see also [14]) are added to *CHF* then conservativity breaks. Intuitively, the reason is that lazy futures may remove some nondeterminism compared to usual futures: While usual futures allow any interleaving of the concurrent evaluation, lazy futures forbid some of them, since their computation cannot start before their value is demanded by some other thread. Since lazy futures can also be implemented in the `unsafeInterleaveIO`-extension of Concurrent Haskell our counterexample implies that Concurrent Haskell with an unrestricted use of `unsafeInterleaveIO` is not safe. Our counterexample does not rely on particulars of the implementation like [9].

As program equivalence for *CHF* we use *contextual equivalence*: two programs are equal iff their observable behavior is indistinguishable even if the programs are plugged as a subprogram into any arbitrary context. Besides observing whether a program can terminate (called *may-convergence*) our notion of contextual equivalence also observes whether a program never loses the ability to terminate after some reductions (called *should-convergence* or sometimes must-convergence, see e.g. [3, 15, 22, 23]). The latter notion slightly differs from the classic notion of must-convergence (e.g. [4]), which additionally requires that all possible computation paths are finite. Some advantages of should-convergence (compared to classical must-convergence) are that restricting the evaluator to *fair scheduling* does not modify the convergence predicates nor contextual equivalence; that equivalence based on may- and should-convergence is invariant under a whole class of test-predicates (see [26]), and inductive reasoning is available as a tool to prove should-convergence.

Results. The lessons learned are that there are declarative and also very expressive pure nonstrict functional languages with a safe extension by IO-monads and concurrency, with valid monad laws, provided `seq`'s first argument is restricted to functional types. Since *CHF* also includes the core parts of Concurrent Haskell our results also imply that Concurrent Haskell conservatively embeds pure Haskell. This also justifies to use well-understood (also denotational) semantics for the pure subcalculus, for example the free theorems in the presence of `seq` [8], or results from call-by-need lambda calculi (e.g. [13, 27]) for reasoning on pure expressions inside Concurrent Haskell. The proof of the main results appears to be impossible by a direct attack. We use the correspondence (see [24]) of the calculus *CHF* with a calculus *CHFI* that unravels recursive bindings into infinite trees and uses call-by-name reduction. In the pure (deterministic) sublanguage *PFI* of *CHFI*, an applicative bisimulation can be shown to be a congruence, using the method of Howe [6, 7, 21], however extended to infinite expressions. This result enables us to prove the main result on infinite expressions, i.e. *CHFI* conservatively extends *PFI*. The final proof step is then to translate the result back to the calculus *CHF* and its pure deterministic sublanguage *PF*.

The **structure** of the paper is as follows. In Section 2 we recall the calculus *CHF* and introduce its pure fragment *PF*. In Section 3 we introduce the two (sub-)calculi *PFI* and *PFMI* with infinite expressions and define applicative bisimulation. In Section 4 we show that bisimulation of *PFI* and *PFMI* coincide and also that contextual equivalence is equivalent to bisimulation in *PFI*. In Section 5 we briefly introduce the process calculus with infinite expressions *CHFI* and show that contextual equivalent expressions of *PFI* are also equivalent in *CHFI*. In Section 6 we go back to the calculi *CHF* and *PF* and prove our Main Theorem 6.4 showing that *CHF* is a conservative extension of *PF*. We show that extending *CHF* by lazy futures breaks conservativity. Finally, we conclude in Section 7. To keep track of the different calculi we summarize some distinguishing properties in the following table:

language	processes	expressions	monadic expressions	sublanguage of
CHF	yes	finite	yes	–
CHFI	yes	infinite	yes	–
PF	no	finite	no	*CHF*
PFI	no	infinite	no	*CHFI*, *PFMI*
PFMI	no	infinite	yes	*CHFI*

2 The CHF-Calculus and its Pure Fragment

We recall the calculus *CHF* modelling Concurrent Haskell with futures [24].

The syntax of CHF consists of processes which have expressions as subterms. Let *Var* be a countably infinite set of variables. We denote variables with x, x_i, y, y_i. *Processes* $Proc_{CHF}$ are generated by the following grammar where $e \in Expr_{CHF}$ is an arbitrary expression (defined below):

$$P, P_i \in Proc_{CHF} ::= P_1 \mid P_2 \;\mid\; \nu x.P \;\mid\; x \Leftarrow e \;\mid\; x = e \;\mid\; x\,\mathbf{m}\,e \;\mid\; x\,\mathbf{m}\,-$$

Parallel composition $P_1 \mid P_2$ constructs concurrently running threads (or other components), *name restriction* $\nu x.P$ restricts the scope of variable x to process P. A *concurrent thread* $x \Leftarrow e$ evaluates the expression e and binds the result of the evaluation to the variable x. The variable x is called the *future* x. In a process there is usually one distinguished thread – the *main thread* – which is labeled with "main" (as notation we use $x \overset{\text{main}}{\Longleftarrow} e$). MVars behave like one place buffers, i.e. if a thread wants to fill an already *filled MVar* $x\,\mathbf{m}\,e$, the thread blocks, and a thread also blocks if it tries to take something from an *empty MVar* $x\,\mathbf{m}\,-$. 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 future, a name of an MVar, or a binding variable. A process is *well-formed*, if all introduced variables are pairwise distinct, and there exists at most one main thread $x \overset{\text{main}}{\Longleftarrow} e$.

We assume a set of *data constructors* c which is partitioned into sets, such that each family represents a type T. The constructors of a type T are ordered, i.e. we write $c_{T,1}, \ldots, c_{T,|T|}$, where $|T|$ is the number of constructors belonging to type T. We omit the index T, i in $c_{T,i}$ if it is clear from the context. Each data constructor $c_{T,i}$ has a fixed arity $\mathrm{ar}(c_{T,i}) \geq 0$. For instance the type `Bool` has constructors `True` and `False` (both of arity 0) and the type `List` has constructors `Nil` (of arity 0) and `Cons` (of arity 2). We assume that there is a unit type `()` with a single constant `()` as constructor.

Expressions $Expr_{CHF}$ and the subset of monadic expressions $MExpr_{CHF}$ are generated by the following grammar:

$$\begin{aligned} e, e_i \in Expr_{CHF} ::=\; & x \mid me \mid \lambda x.e \mid (e_1\ e_2) \mid c\ e_1 \ldots e_{\mathrm{ar}(c)} \mid \texttt{seq}\ e_1\ e_2 \\ & \mid \texttt{case}_T\ e\ \texttt{of}\ (c_{T,1}\ x_1 \ldots x_{\mathrm{ar}(c_{T,1})} \to e_1) \ldots (c_{T,|T|}\ x_1 \ldots x_{\mathrm{ar}(c_{T,|T|})} \to e_{|T|}) \\ & \mid \texttt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e \end{aligned}$$

$$\begin{aligned} me \in MExpr_{CHF} ::=\; & \texttt{return}\ e \mid e_1\ \texttt{>>=}\ e_2 \mid \texttt{future}\ e \mid \texttt{takeMVar}\ e \\ & \mid \texttt{newMVar}\ e \mid \texttt{putMVar}\ e_1\ e_2 \end{aligned}$$

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

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

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

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

$$\frac{\Gamma\vdash e_1::\mathtt{IO}\ \tau_1,\Gamma\vdash e_2::\tau_1\to\mathtt{IO}\ \tau_2}{\Gamma\vdash e_1\,\mathtt{>>=}\,e_2::\mathtt{IO}\ \tau_2}\quad \frac{\forall i:\Gamma(x_i)=\tau_i,\ \forall i:\Gamma\vdash e_i::\tau_i,\ \Gamma\vdash e::\tau}{\Gamma\vdash(\mathtt{letrec}\ x_1=e_1,\ \ldots\ x_n=e_n\ \mathtt{in}\ e)::\tau}$$

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

$$\frac{\Gamma\vdash e::\tau_1\ \text{ and }\tau_1=(T\ \ldots),\ \forall i:\Gamma\vdash(c_{T,i}\ x_{i,1}\ \ldots\ x_{i,n_i})::\tau_1,\ \forall i:\Gamma\vdash e_i::\tau_2}{\Gamma\vdash(\mathtt{case}_T\ e\ \mathtt{of}(c_{T,1}\ x_{1,1}\ \ldots\ x_{1,n_1}\to e_1)\ldots(c_{T,|T|}\ x_{|T|,1}\ \ldots\ x_{|T|,n_{|T|}}\to e_{|T|}))::\tau_2}$$

Fig. 1. Typing rules

Besides the usual constructs of the lambda calculus (variables, abstractions, applications) expressions comprise *constructor applications* $(c\ e_1\ldots e_{\mathrm{ar}(c)})$, `case`-*expressions* for deconstruction, `seq`-expressions for sequential evaluation, `letrec`-*expressions* to express recursive shared bindings and monadic expressions which allow to form monadic actions.

For `case`-expressions there is a $\mathtt{case}_T$-construct for every type T and there is a `case`-alternative for every constructor of type T. The variables in a `case`-pattern $(c\ x_1\ \ldots\ x_{\mathrm{ar}(c)})$ and also the bound variables in a `letrec`-expression must be pairwise distinct. We sometimes abbreviate the `case`-alternatives as *alts*, $\mathtt{case}_T\ e\ \mathtt{of}\ alts$. The expression `return` e is the monadic action which returns e as result, the operator `>>=` allows one to combine monadic actions, the expression `future` e will create a concurrent thread evaluating the action e, the operation `newMVar` e will create an MVar filled with e, `takeMVar` x will return the content of MVar x, and `putMVar` $x\ e$ will fill MVar x with content e.

Variable binders are introduced by abstractions, `letrec`-expressions, `case`-alternatives, and for processes by the restriction $\nu x.P$. For the induced notion of of free and bound variables we use $FV(P)$ ($FV(e)$, resp) to denote the free variables of process P (expression e, resp.) and $=_\alpha$ to denote α-equivalence. We use *distinct variable convention*, i.e. all free variables are distinct from bound variables, all bound variables are pairwise distinct, and reductions implicitly perform α-renaming to obey this convention. For processes *structural congruence* $\equiv$ is defined as the least congruence satisfying the equations: $P_1\mid P_2\equiv P_2\mid P_1$; $\nu x_1.\nu x_2.P\equiv\nu x_2.\nu x_1.P$; $(P_1\mid P_2)\mid P_3\equiv P_1\mid(P_2\mid P_3)$; $P_1\equiv P_2$, if $P_1=_\alpha P_2$; and $(\nu x.P_1)\mid P_2\equiv\nu x.(P_1\mid P_2)$, if $x\notin FV(P_2)$.

Monadic Computations

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

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

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

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

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

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

Functional Evaluation

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

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

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

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

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

Fig. 2. Call-by-name reduction rules of *CHF*

We use a monomorphic type system where data constructors and monadic operators are treated like "overloaded" polymorphic constants. The syntax of types Typ_{CHF} is $\tau, \tau_i \in Typ_{CHF} ::= \texttt{IO}\ \tau \mid (T\ \tau_1\ \ldots\ \tau_n) \mid \texttt{MVar}\ \tau \mid \tau_1 \to \tau_2$. Here $\texttt{IO}\ \tau$ means that an expression of type τ is the result of a monadic action, $\texttt{MVar}\ \tau$ stands for an `MVar`-reference with content type τ, and $\tau_1 \to \tau_2$ is a function type. With $\texttt{types}(c)$ we denote the set of monomorphic types of constructor c. To fix the types during reduction, we assume that every variable has a fixed (built-in) type: Let Γ be the global typing function for variables, i.e. $\Gamma(x)$ is the type of variable x. We use the notation $\Gamma \vdash e :: \tau$ to express that τ can be derived for expression e using the global typing function Γ. For processes $\Gamma \vdash P :: \texttt{wt}$ means that the process P can be well-typed using the global typing function Γ. The typing rules are given in Fig. 1. Note that the first argument of `seq` must not be an `IO`- or `MVar`-type, since otherwise the monad laws would not hold in *CHF* (and even not in Haskell, see [24]). 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.

2.1 Operational Semantics and Program Equivalence

In [24] a call-by-need as well as a call-by-name small step reduction for CHF were introduced and it has been proved that both reduction strategies induce the same notion of program equivalence. Here we will only recall the call-by-name reduction. As a first step we introduce some classes of contexts.

On the process level the *process contexts PCtxt* are defined as follows, where $P \in Proc_{CHF}$: $\mathbb{D}, \mathbb{D}_i \in PCtxt ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}$.

On expressions usual (call-by-name) *expression evaluation contexts ECtxt* are defined by: $\mathbb{E}, \mathbb{E}_i \in \mathit{ECtxt} ::= [\cdot] \mid (\mathbb{E}\ e) \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ \mathit{alts}) \mid (\texttt{seq}\ \mathbb{E}\ e)$.

To enforce the evaluation of the (first) argument of the monadic operators `takeMVar` and `putMVar` the class of *forcing contexts FCtxt* are required, which are defined as follows: $\mathbb{F}, \mathbb{F}_i \in \mathit{FCtxt} ::= \mathbb{E} \mid (\texttt{takeMVar}\ \mathbb{E}) \mid (\texttt{putMVar}\ \mathbb{E}\ e)$.

A *functional value* is an abstraction or a constructor application, a *value* is a functional value or a monadic expression of *MExpr*.

Definition 2.1 (Call-by-name Standard Reduction). *The call-by-name standard reduction $\xrightarrow{CHF}$ is defined by the rules in Fig. 2 where we additionally assume that $\xrightarrow{CHF}$ is closed w.r.t. PCtxt-contexts and structural congruence, i.e. if $P \equiv \mathbb{D}[P']$, $Q \equiv \mathbb{D}[Q']$ and $P' \xrightarrow{CHF} Q'$ then also $P \xrightarrow{CHF} Q$. We also assume that only well-formed processes are reducible.*

The rules for functional evaluation include classical call-by-name β-reduction (rule (beta)), a rule for copying shared bindings into a needed position (rule (cpce)), rules to evaluate `case`- and `seq`-expressions (rules (case) and (seq)), and the rule (mkbinds) to move `letrec`-bindings into the global set of shared bindings. We now explain the rules for monadic computation: The rule (lunit) is the direct implementation of the monad and applies the first monad law to proceed a sequence of monadic actions. The rules (nmvar), (tmvar), and (pmvar) handle the MVar creation and access. Note that a `takeMVar`-operation can only be performed on a filled MVar, and a `putMVar`-operation needs an empty MVar for being executed. The rule (fork) spawns a new concurrent thread, where the calling thread receives the name of the thread (the future) as result. If a concurrent thread finished its computation, then the result is shared as a global binding and the thread is removed (rule (unIO)). Note that if the calling thread needs the result of the future, it gets blocked until the result becomes available.

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. Thereby the usual observation is whether the evaluation of the process successfully terminates or not. In nondeterministic (and also concurrent) calculi this observation is called may-convergence, and it does *not* suffice 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 is called should-convergence (see [22–24]).

Definition 2.2. *A process P is* successful *iff it is well-formed and contains a main thread of the form $x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ e$.*

A process P may-converges *(written as $P{\downarrow}_{CHF}$), iff it is well-formed and reduces to a successful process, i.e. $\exists P' : P \xrightarrow{CHF,*} P' \wedge P'$ is successful. If $P{\downarrow}_{CHF}$ does not hold, then P* must-diverges *written as $P{\Uparrow}_{CHF}$.*

A process P should-converges *(written as $P{\Downarrow}_{CHF}$), iff it is well-formed and remains may-convergent under reduction, i.e. $\forall P' : P \xrightarrow{CHF,*} P' \implies P'{\downarrow}_{CHF}$. If P is not should-convergent then we say P* may-diverges *written as $P{\uparrow}_{CHF}$.*

Note that a process P is may-divergent if there is a finite reduction sequence $P \xrightarrow{CHF,*} P'$ such that $P'{\Uparrow_{CHF}}$. We sometimes write $P{\downarrow_{CHF}}P'$ (or $P{\uparrow_{CHF}}P'$, resp.) if $P \xrightarrow{CHF,*} P'$ and P' is a successful (or must-divergent, resp.) process.

Definition 2.3. Contextual approximation $\leq_{c,CHF}$ *and* contextual equivalence $\sim_{c,CHF}$ *on processes are defined as* $\leq_{c,CHF}:=\leq_{\downarrow_{CHF}} \cap \leq_{\Downarrow_{CHF}}$ *and* $\sim_{c,CHF}:=\leq_{c,CHF} \cap \geq_{c,CHF}$ *where for* $\chi \in \{\downarrow_{CHF}, \Downarrow_{CHF}\}$*:*

$$P_1 \leq_\chi P_2 \;\; \textit{iff} \;\; \forall \mathbb{D} \in \mathit{PCtxt} : \mathbb{D}[P_1]\chi \implies \mathbb{D}[P_2]\chi$$

Let $\mathbb{C} \in \mathit{Ctxt}$ *be contexts that are constructed by replacing a subexpression in a process by a (typed) context hole. Contextual approximation* $\leq_{c,CHF}$ *and contextual equivalence* $\sim_{c,CHF}$ *on equally typed expressions are defined as* $\leq_{c,CHF}:= \leq_{\downarrow_{CHF}} \cap \leq_{\Downarrow_{CHF}}$ *and* $\sim_{c,CHF}:= \leq_{c,CHF} \cap \geq_{c,CHF}$*, where for expressions* e_1, e_2 *of type* τ *and* $\chi \in \{\downarrow_{CHF}, \Downarrow_{CHF}\}$*:* $e_1 \leq_\chi e_2$ *iff* $\forall \mathbb{C}[\cdot^\tau] \in \mathit{Ctxt} : \mathbb{C}[e_1]\chi \implies \mathbb{C}[e_2]\chi$.

2.2 The Pure Fragment *PF* of *CHF*

The calculus *PF* comprises the pure (i.e. non-monadic) expressions and types of *CHF*, i.e. expressions Expr_{PF} are built according to the grammar:

$$\begin{array}{rl} e, e_i \in \mathit{Expr}_{PF} ::= & x \mid \lambda x.e \mid (e_1\ e_2) \mid c\ e_1 \ldots e_{\mathrm{ar}(c)} \mid \mathtt{seq}\ e_1\ e_2 \\ \mid & \mathtt{case}_T\ e\ \mathtt{of}\ (c_{T,1}\ x_1 \ldots x_{\mathrm{ar}(c_{T,1})} \to e_1) \ldots (c_{T,|T|}\ x_1 \ldots x_{\mathrm{ar}(c_{T,|T|})} \to e_{|T|}) \\ \mid & \mathtt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \mathtt{in}\ e \end{array}$$

The calculus *PF* only has *pure types* $\mathit{Typ}_P \subset \mathit{Typ}_{CHF}$ according to the following grammar where T is a type-constructor: $\tau, \tau_i \in \mathit{Typ}_P ::= (T\ \tau_1\ \ldots\ \tau_n) \mid \tau_1 \to \tau_2$.

An expression $e \in \mathit{Expr}_{PF}$ is *well-typed with type* $\tau \in \mathit{Typ}_P$ iff $\Gamma \vdash e :: \tau$ can be derived by the typing rules of Fig. 1.

Instead of providing an operational semantics inside the expressions of *PF*, we define convergence of Expr_{PF} by using the (larger) calculus *CHF* as follows: A *PF*-expression e *converges* (denoted by $e{\downarrow_{PF}}$) iff $y \overset{\mathsf{main}}{\Longleftarrow} \mathtt{seq}\ e\ (\mathtt{return}\ ()){\downarrow_{CHF}}$ for some $y \notin FV(e)$. The results in [24] show that convergence does not change if we would have used call-by-need evaluation in CHF (defined in [24]). This allows one to show that *PF* is semantically equivalent (w.r.t. contextual equivalence) to a usual extended call-by-need `letrec`-calculus as e.g. the calculi in [28, 27].

PF-contexts Ctxt_{PF} are Expr_{PF}-expressions where a subterm is replaced by the context hole. For $e_1, e_2 \in \mathit{Expr}_{PF}$ of type τ, the relation $e_1 \leq_{c,PF} e_2$ holds, if for all $\mathbb{C}[\cdot_\tau] \in \mathit{Ctxt}_{PF}$, $\mathbb{C}[e_1]{\downarrow_{PF}} \implies \mathbb{C}[e_2]{\downarrow_{PF}}$. Note that it is not necessary to observe should-convergence, since the calculus *PF* is deterministic.

Our main goal of this paper is to show that for any $e_1, e_2 :: \tau \in \mathit{Expr}_{PF}$ the following holds: $e_1 \sim_{c,PF} e_2 \implies e_1 \sim_{c,CHF} e_2$. This implies that two equal pure expressions cannot be distinguished in the concurrent calculus with futures.

3 Simulation in the Calculi of Infinite Expressions

We will now consider a simulation relation in two variants of *PF* which use *infinite* expressions. We will first introduce two calculi *PFI* and *PFMI* with infinite expressions and then define similarity for both calculi. Using Howe's method it is possible to show that both similarities are precongruences, for space reasons the congruence proof can be found in the appendix. To distinguish infinite expressions from finite expressions (on the meta-level) we always use e, e_i for finite expressions and r, s, t for infinite expressions. Nevertheless, in abuse of notation we will use the same meta symbols for finite as well as infinite contexts.

Definition 3.1. *The language of PFMI is defined as follows: PFMI uses the same types Typ_{CHF} as the calculus CHF.* Infinite expressions *$IExpr_{PFMI}$ of the calculus PFMI are defined like expressions $Expr_{CHF}$ omitting the letrec-component, adding a constant* `Bot` *and constants for names of* `MVar`*s, and interpreting the grammar* coinductively*, i.e. the grammar is as follows*

$$\begin{array}{l} r,s,t \in \mathit{IExpr}_{\mathit{PFMI}} ::= x \mid a \mid ms \mid \texttt{Bot} \mid \lambda x.s \mid (s_1\ s_2) \\ \quad \mid (c\ s_1 \cdots s_{\mathrm{ar}(c)}) \mid \texttt{seq}\ s_1\ s_2 \\ \quad \mid \texttt{case}_T\ s\ \texttt{of}\ (c_{T,1}\ x_1 \cdots x_{\mathrm{ar}(c_{T,1})} \to s_1) \ldots (c_{T,|T|}\ x_1 \cdots x_{\mathrm{ar}(c_{T,|T|})} \to s_{|T|}) \end{array}$$

$$\begin{array}{l} ms \in \mathit{IMExpr}_{\mathit{PFMI}} ::= \texttt{return}\ s \mid s_1\ \texttt{>>=}\ s_2 \mid \texttt{future}\ s \mid \texttt{takeMVar}\ s \\ \quad \mid \texttt{newMVar}\ s \mid \texttt{putMVar}\ s_1\ s_2 \end{array}$$

where c are data constructors and a are from an infinite set of 0-ary constants of type `MVar` *τ for every τ. An infinite expression $s \in IExpr_{PFMI}$ is* well-typed *(with type τ) iff $\Gamma \vdash s :: \tau$ by the typing rules in Fig. 1 where only the rules for expressions are used and the rules are applied coinductively over the expression syntax. Additionally the typing rules include the axioms*

$\Gamma \vdash a :: \texttt{MVar}\ \tau$ *if constant a is of type* $\texttt{MVar}\ \tau$ $\qquad \Gamma \vdash \texttt{Bot} :: \tau$

Note that types are still defined inductively. Hence, infinite types are not allowed and well-typed infinite expressions must be typeable by a finite type.

Definition 3.2. *The language of PFI is a sublanguage of PFMI by omitting* `IO`*- and* `MVar`*-types and monadic operators: PFI uses pure types Typ_P as types and* infinite expressions *$IExpr_{PFI}$ of the calculus PFI are defined like expressions $Expr_{PFMI}$ by omitting several possibilities, and interpreting the grammar coinductively, i.e. the grammar is as follows*

$$\begin{array}{l} r,s,t \in \mathit{IExpr}_{\mathit{PFI}} ::= x \mid \texttt{Bot} \mid \lambda x.s \mid (s_1\ s_2) \mid (c\ s_1 \ldots s_{\mathrm{ar}(c)}) \mid \texttt{seq}\ s_1\ s_2 \\ \quad \mid \texttt{case}_T\ s\ \texttt{of}\ (c_{T,1}\ x_1 \ldots x_{\mathrm{ar}(c_{T,1})} \to s_1) \ldots (c_{T,|T|}\ x_1 \ldots x_{\mathrm{ar}(c_{T,|T|})} \to s_{|T|}) \end{array}$$

An infinite expression $s \in Expr_{PFI}$ is well-typed with type *$\tau \in Typ_P$ iff $\Gamma \vdash s :: \tau$ can be derived by coinductively applying the typing rules for expressions of Fig. 1 and the axiom $\Gamma \vdash$* `Bot` *$:: \tau$.*

(beta) $\mathbb{E}[((\lambda x.s_1)\ s_2)] \to \mathbb{E}[s_1[s_2/x]]$
(case) $\mathbb{E}[\texttt{case}_T\ (c\ s_1\ \dots\ s_n)\ \texttt{of}\ \dots((c\ y_1\ \dots\ y_n) \to s)\dots] \to \mathbb{E}[s[s_1/y_1,\dots,s_n/y_n]]$
(seq) $\mathbb{E}[(\texttt{seq}\ v\ s)] \to \mathbb{E}[s]$ if v is a functional value

Fig. 3. Call-by-name reduction rules on infinite expressions

In both calculi a *functional value* is an abstraction or a constructor application (except for the constant `Bot`), and a *value* is a functional value or a monadic expression of $\mathit{IMExpr}_{\mathit{PFMI}}$ in the case of the calculus *PFMI*. With $\mathit{IExpr}^c_{\mathit{PFMI}}$ ($\mathit{IExpr}^c_{\mathit{PFI}}$, resp.) we denote the set of *closed* infinite expressions.

We now define the operational semantics for both calculi. In abuse of notation we sometimes use a single meta-symbol which is implicitly parametrized by *PFMI* or *PFI*. The (infinite) call-by-name evaluation contexts *IECtxt* are defined by the following (inductively interpreted) grammar, where $s \in \mathit{IExpr}_{\mathit{PFMI}}$ (or $s \in \mathit{IExpr}_{\mathit{PFI}}$ resp.): $\mathbb{E}, \mathbb{E}_i \in \mathit{IECtxt} ::= [\cdot] \mid (\mathbb{E}\ s) \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ \mathit{alts}) \mid (\texttt{seq}\ \mathbb{E}\ s)$. The (call-by-name) reduction rules on infinite expressions are defined in Fig. 3. Note that the substitutions used in (beta) and (case) may substitute infinitely many occurrences of variables. For *PFMI* reduction cannot extract subexpressions from monadic expressions, hence they behave similarly to constants.

The (normal-order) call-by-name reduction is written $s \xrightarrow{PFMI} t$ (s $\xrightarrow{PFI} t$, resp.), and $s{\downarrow_{PFMI}}t$ ($s{\downarrow_{PFI}}t$, resp.) means that there is a value t, such that $s \xrightarrow{PFMI,*} t$ ($s \xrightarrow{PFI,*} t$). If we are not interested in the specific value t we also write $s{\downarrow_{PFMI}}$ (or $s{\downarrow_{PFI}}$, resp.).

We define similarity for both calculi *PFMI* and *PFI*. For simplicity, we sometimes use as e.g. in [6] the higher-order abstract syntax and write $\xi(..)$ for an expression with top operator ξ, which may be all possible term constructors, like `case`, application, a constructor, `seq`, or λ, and θ for an operator that may be the head of a value, i.e. a constructor or monadic operator or λ. Note that ξ and θ may represent also the binding λ using $\lambda(x.s)$ as representing $\lambda x.s$. In order to stick to terms, and be consistent with other papers like [6], we assume that removing the top constructor $\lambda x.$ in relations is done after a renaming. For example, $\lambda x.s\ \mu\ \lambda y.t$ is renamed before further treatment to $\lambda z.s[z/x]\ \mu\ \lambda z.t[z/y]$ for a fresh variable z. Hence $\lambda x.s\ \mu\ \lambda x.t$ means $s\ \mu^o\ t$ for open expressions s, t, if μ is a relation on closed expressions. Similarly for `case`, where the first argument is without scope, and the case alternative like $(c\ x_1 \dots x_n \to s)$ is seen as s with a scoping of $x_1, \dots x_n$. We assume that binary relations η relate expressions of equal type. A substitution σ that replaces all free variables by closed infinite expressions is called a *closing substitution.*

Definition 3.3. *Let η be a binary relation on closed infinite expressions. Then the* open extension *η^o on all infinite expressions is defined as $s\ \eta^o\ t$ for any expressions s, t iff for all closing substitutions σ: $\sigma(s)\ \eta\ \sigma(t)$. Conversely, for binary relations μ on open expressions, $(\mu)^c$ is the restriction to closed expressions.*

Lemma 3.4. *For a relation η on closed expressions, the equation $((\eta)^o)^c = \eta$ holds, and $s\ \eta^o\ t$ implies $\sigma(s)\ \eta^o\ \sigma(t)$ for any substitution σ. For a relation*

μ on open expressions the inclusion $\mu \subseteq ((\mu)^c)^o$ is equivalent to $s\ \mu\ t \implies \sigma(s)\ (\mu)^c\ \sigma(t)$ for all closing substitutions σ.

Definition 3.5. *Let $\leq_{b,PFMI}$ (called* similarity*) be the greatest fixpoint, on the set of binary relations over closed (infinite) expressions, of the following operator F_{PFMI} on binary relations η over closed expressions $IExpr^c_{PFMI}$:*

For $s, t \in IExpr^c_{PFMI}$ the relation $s\ F_{PFMI}(\eta)\ t$ holds iff $s{\downarrow}_{PFMI}\theta(s_1,\dots,s_n)$ implies that there exist $t_1,\dots,t_n$ such that $t{\downarrow}_{PFMI}\theta(t_1,\dots,t_n)$ and $s_i\ \eta^o\ t_i$ for $i = 1,\dots,n$.

The operator F_{PFMI} is monotone, hence the greatest fixpoint $\leq_{b,PFMI}$ exists.

Proposition 3.6 (Coinduction). *The principle of coinduction for the greatest fixpoint of F_{PFMI} shows that for every relation η on closed expressions with $\eta \subseteq F_{PFMI}(\eta)$, we derive $\eta \subseteq \leq_{b,PFMI}$. This also implies $(\eta)^o \subseteq (\leq_{b,PFMI})^o$.*

Similarly, Definition 3.5 and Proposition 3.6 can be transferred to *PFI*, where we use $\leq_{b,PFI}$ and F_{PFI} as notation. Determinism of $\xrightarrow{PFMI}$ implies:

Lemma 3.7. *If $s \xrightarrow{PFMI} s'$, then $s' \leq^o_{b,PFMI} s$ and $s \leq^o_{b,PFMI} s'$.*

In the appendix (Theorem A.16) we show that $\leq^o_{b,PFMI}$ and $\leq^o_{b,PFI}$ are precongruences by adapting Howe's method [6, 7] to the infinite syntax of the calculi.

Theorem 3.8. *$\leq^o_{b,PFMI}$ is a precongruence on infinite expressions $IExpr_{PFMI}$. If σ is a substitution, then $s \leq^o_{b,PFMI} t$ implies $\sigma(s) \leq^o_{b,PFMI} \sigma(t)$.*

$\leq^o_{b,PFI}$ is a precongruence on infinite expressions $IExpr_{PFI}$. If σ is a substitution, then $s \leq^o_{b,PFI} t$ implies $\sigma(s) \leq^o_{b,PFI} \sigma(t)$.

4 Behavioral and Contextual Preorder in *PFI* and *PFMI*

In this section we investigate the relationships between the behavioral and contextual preorders in the two calculi *PFI* and *PFMI* of infinite expressions.

We know that $\leq^o_{b,PFI}$ as well as $\leq^o_{b,PFMI}$ are precongruences. We will show below that $\leq^o_{b,PFMI}$ is a conservative extension of $\leq^o_{b,PFI}$, which is not obvious, since the $\leq_{b,PFMI}$-test for abstractions has to take into account more arguments than the $\leq_{b,PFI}$-test. First we will show that in *PFI*, the contextual and behavioral preorder coincide. Note that this is wrong for *PFMI*, because there are expressions like `return True` and `return False` that cannot be contextually distinguished since *PFMI* cannot look into the components of these terms.

4.1 Behavioral and Contextual Preorder in *PFI*

In this subsection we treat the properties of the pure functional language *PFI* with infinite expressions. Let $ICtxt_{PFI}$ be the set of all contexts in *PFI*.

Definition 4.1. *The contextual equivalence w.r.t. PFI is defined as* $\sim_{c,PFI} := \leq_{c,PFI} \cap \geq_{c,PFI}$ *where for equally typed expressions* $s, t :: \tau$
$s \leq_{c,PFI} t$ *iff* $\forall \mathbb{C}[\cdot :: \tau] \in \mathit{ICtxt}_{PFI} : \mathbb{C}[s]{\downarrow_{PFI}} \implies \mathbb{C}[t]{\downarrow_{PFI}}$.

Lemma 4.2. $\leq^o_{b,PFI} \;\subseteq\; \leq_{c,PFI}$.

Proof. Let s, t be expressions with $s \leq^o_{b,PFI} t$ such that $\mathbb{C}[s]{\downarrow_{PFI}}$. Let σ be a substitution that replaces all free variables of $\mathbb{C}[s], \mathbb{C}[t]$ by `Bot`. The properties of the call-by-name reduction show that also $\sigma(\mathbb{C}[s]){\downarrow_{PFI}}$. Since $\sigma(\mathbb{C}[s]) = \sigma(\mathbb{C})[\sigma(s)]$, $\sigma(\mathbb{C}[t]) = \sigma(\mathbb{C})[\sigma(t)]$ and since $\sigma(s) \leq^o_{b,PFI} \sigma(t)$, we obtain from the precongruence property of $\leq^o_{b,PFI}$ that also $\sigma(\mathbb{C}[s]) \leq_{b,PFI} \sigma(\mathbb{C}[t])$. Hence $\sigma(\mathbb{C}[t]){\downarrow_{PFI}}$. This is equivalent to $\mathbb{C}[t]{\downarrow_{PFI}}$, since free variables are replaced by `Bot`, and thus they cannot overlap with redexes. Hence $\leq^o_{b,PFI} \;\subseteq\; \leq_{c,PFI}$.

Lemma 4.3. *In PFI, the contextual preorder on expressions is contained in the behavioral preorder on open expressions, i.e.* $\leq_{c,PFI} \;\subseteq\; \leq^o_{b,PFI}$.

Proof. We show that $\leq^c_{c,PFI}$ satisfies the fixpoint condition, i.e. $\leq^c_{c,PFI} \subseteq F_{PFI}(\leq^c_{c,PFI})$: Let s, t be closed and $s \leq_{c,PFI} t$. If $s{\downarrow_{PFI}}\theta(s_1, \ldots, s_n)$, then also $t{\downarrow_{PFI}}$. Using the appropriate `case`-expressions as contexts, it is easy to see that $t{\downarrow_{PFI}}\theta(t_1, \ldots, t_n)$. Now we have to show that $s_i \leq^c_{c,PFI} t_i$. This could be done using an appropriate context $\mathbb{C}_i$ that selects the components, i.e. $\mathbb{C}_i[s] \xrightarrow{PFI,*} s_i$ and $\mathbb{C}_i[t] \xrightarrow{PFI,*} t_i$ Since reduction preserves similarity and Lemma 4.2 show that $r \xrightarrow{PFI} r'$ implies $r \leq_{c,PFI} r'$ holds. Moreover, since $\leq^o_{c,PFI}$ is obviously a precongruence, we obtain that $s_i \leq^o_{c,PFI} t_i$. Thus the proof is finished.

Concluding, Lemmas 4.2 and 4.3 imply:

Theorem 4.4. *In PFI the behavioral preorder is the same as the contextual preorder on expressions, i.e.* $\leq^o_{b,PFI} \;=\; \leq_{c,PFI}$.

In the proofs in Section 5 for the language *PFMI* the notion of recursive replacement and a technical lemma on $\leq^o_{b,PFI}$ are required.

Definition 4.5. *Let x be a variable and s be a PFMI-expression (there may be free occurrences of x in s) of the same type. Then $s \,/\!/\, x$ is a substitution that replaces recursively x by s. In case s is the variable x, then $s \,/\!/\, x$ is the substitution $x \mapsto$* `Bot`.

For example, $(a\ x) \,/\!/\, x$ replaces x by the infinite expression $(a\ (a\ (a\ \ldots)))$.

In the appendix (Lemma A.18) we prove the following lemma:

Lemma 4.6. *Let x be a variable and s_1, s_2, t_1, t_2 be PFMI-expressions with $s_i \leq^o_{b,PFMI} t_i$ for $i = 1, 2$. Then $s_2[s_1 \,/\!/\, x] \leq^o_{b,PFMI} t_2[t_1 \,/\!/\, x]$.*

4.2 Behavioral Preorder in *PFMI*

The goal is to show that for *PFI*-expressions s, t, the behavioral preorders w.r.t. *PFMI* and *PFI* are equivalent, i.e., that $\leq_{b,PFMI}$ is a conservative extension of $\leq_{b,PFI}$ when extending the language *PFI* to *PFMI*. This is not immediate, since the behavioral preorders w.r.t. *PFMI* requires to test abstractions on more closed expressions than *PFI*. Put differently, the open extension of relations is w.r.t. a larger set of closing substitutions.

Definition 4.7. *Let $\phi : PFMI \to PFI$ be the mapping with $\phi(x) := x$, if x is a variable; $\phi(c\ s_1 \dots s_n) := ()$, if c is a monadic operator; $\phi(a) := ()$, if a is a name of an MVar; and $\phi(\xi(s_1, \dots, s_n)) := \xi(\phi(s_1), \dots, \phi(s_n))$ for any other operator ξ. Also the types are translated by replacing all $(\mathtt{IO}\ \tau)$ and $(\mathtt{MVar}\ \tau)$-types by type $()$ and retaining the other types.*

This translation is *compositional*, i.e., it translates along the structure: $\phi(\mathbb{C}[s]) = \phi(\mathbb{C})[\phi(s)]$ if $\phi(\mathbb{C})$ is again a context, or $\phi(\mathbb{C}[s]) = \phi(\mathbb{C})$ if the hole of the context is removed by the translation. In the following we write $\phi(\mathbb{C})[\phi(s)]$ also in the case that the hole is removed, in which case we let $\phi(\mathbb{C})$ be a constant function. Now the following lemma is easy to verify:

Lemma 4.8. *For all closed PFMI-expressions s it holds: $s{\downarrow}_{PFMI}$ iff $\phi(s){\downarrow}_{PFI}$, and if $s{\downarrow}_{PFMI}\theta(s_1, \dots, s_n)$ then $\phi(s){\downarrow}_{PFI}\phi(\theta(s_1, \dots, s_n))$. Conversely, if $\phi(s){\downarrow}_{PFI}\theta(s_1, \dots, s_n)$, then $s{\downarrow}_{PFMI}\theta(s'_1, \dots, s'_n)$ such that $\phi(s'_i) = s_i$ for all i.*

Now we show that $\leq_{b,PFI}$ is the same as $\leq_{b,PFMI}$ restricted to *PFI*-expressions using coinduction:

Lemma 4.9. $\leq_{b,PFI}\ \subseteq\ \leq_{b,PFMI}$.

Proof. Let ρ be the relation $\{(s,t) \mid \phi(s) \leq_{b,PFI} \phi(t)\}$ on closed *PFMI*-expressions, i.e., $s\ \rho\ t$ holds iff $\phi(s) \leq_{b,PFI} \phi(t)$. We show that $\rho \subseteq F_{PFMI}(\rho)$. Assume $s\ \rho\ t$ for $s, t \in \mathit{IExpr}_{PFMI}$. Then $\phi(s) \leq_{b,PFI} \phi(t)$. If $\phi(s){\downarrow}_{PFI}\theta(s_1, \dots, s_n)$, then also $\phi(t){\downarrow}_{PFI}\theta(t_1, \dots, t_n)$ and $s_i \leq^o_{b,PFI} t_i$. Now let σ be a *PFMI*-substitution such that $\sigma(s_i), \sigma(t_i)$ are closed. Then $\phi(\sigma)$ is a *PFI*-substitution, hence $\phi(\sigma)(s_i) \leq_{b,PFI} \phi(\sigma)(t_i)$. We also have $\phi(\sigma(s_i)) = \phi(\sigma)(s_i)$, $\phi(\sigma(t_i)) = \phi(\sigma)(t_i)$, since s_i, t_i are *PFI*-expressions and since ϕ is compositional. The relation $s_i\ \rho^o\ t_i$ w.r.t. *PFMI* is equivalent to $\sigma(s_i)\ \rho\ \sigma(t_i)$ for all closing *PFMI*-substitutions σ, which in turn is equivalent $\phi(\sigma(s_i)) \leq_{b,PFI} \phi(\sigma(s_i))$. Hence $s_i\ \rho^o\ t_i$ for all i where the open extension is w.r.t. *PFMI*. Thus $\rho \subseteq F_{PFMI}(\rho)$ and hence $\rho\ \subseteq\ \leq_{b,PFMI}$. Since $\leq_{b,PFI}\ \subseteq\ \rho$, this implies $\leq_{b,PFI}\ \subseteq\ \leq_{b,PFMI}$.

Proposition 4.10. *Let $s, t \in \mathit{IExpr}_{PFI}$. Then $s \leq_{b,PFI} t$ iff $s \leq_{b,PFMI} t$.*

Proof. The relation $s \leq_{b,PFMI} t$ implies $s \leq_{b,PFI} t$, since the fixpoint w.r.t. F_{PFMI} is a subset of the fixpoint of F_{PFI}. The other direction is Lemma 4.9.

Proposition 4.11. *Let x be a variable of type $(\mathtt{MVar}\ \tau)$ for some τ, and let s be a PFMI-expression of the same type such that $x \leq^o_{b,PFMI} s$. Then $s{\downarrow}_{PFMI}x$.*

Proof. Let σ be a substitution such that $\sigma(x) = a$ where a is a name of an MVar, a does not occur in s, $\sigma(s)$ is closed and such that $\sigma(x) \leq_{b,PFMI} \sigma(s)$. We can choose σ in such a way that $\sigma(y)$ does not contain a for any variable $y \neq x$. By the properties of $\leq_{b,PFMI}$, we obtain $\sigma(s){\downarrow_{PFMI}} a$. Since the reduction rules of *PFMI* cannot distinguish between a or x, and since $\sigma(y)$ does not contain a, the only possibility is that s reduces to x.

5 Contextual Equivalence in the Process Calculus *CHFI*

The calculus *CHFI* is the variant of the calculus *CHF* on infinite expressions (see [24]). *CHFI* is similar to *CHF* where instead of finite expressions $Expr_{CHF}$ infinite expressions $IExpr_{PFMI}$ are used, and shared bindings are omitted: *Infinite processes* $IProc_{CHFI}$ are defined by the (inductively interpreted) grammar:

$$S, S_i, \in IProc_{CHFI} ::= S_1 \mid S_2 \ \mid\ x \Leftarrow s \ \mid\ \nu x.S \ \mid\quad \mid\ x\,\mathbf{m}\,s \ \mid\ x\,\mathbf{m}\,- \ \mid\ \mathbf{0}$$

where $s \in IExpr_{PFMI}$ is an infinite expression. $\mathbf{0}$ is the $\mathbf{0}$-process which does nothing. Functional values and values are defined as in the calculus *PFMI*. Typing is according to Fig. 1 where the derivation rules are applied coinductively to infinite expressions. We also use structural congruence $\equiv$ for $IProc_{CHFI}$-processes which is defined in the obvious way where $S \mid \mathbf{0} \equiv S$ is an additional rule.

The standard reduction $\xrightarrow{CHFI}$ of the calculus *CHFI* uses the call-by-name reduction of *PFMI* for expressions (where the monadic operators are executed). For space reasons we do not list all the reduction rules again, they are analogous to rules for *CHF* (see Fig. 2), but work on infinite expressions (and adapted contexts) with the following modifications: For the functional evaluation only the rules (case), (beta), and (seq) are used (since there are no bindings in *CHFI*). The monadic reductions are as in *CHF* except for the (unIO) rule which is replaced by the following variant, where // means the infinite recursive replacement of s for y:

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

We assume reduction to be closed w.r.t. structural congruence $\equiv$ and process contexts, i.e. iff $S_1 \equiv \mathbb{D}[S_1'], S_2 \equiv \mathbb{D}[S_2']$ and $S_1' \xrightarrow{CHFI} S_2'$ then also $S_1 \xrightarrow{CHFI} S_2$.

An infinite process S is successful if it is well-formed (i.e. all introduced variables are distinct) and if it is of the form $S \equiv \nu x_1, \ldots, x_n.(x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ s \mid S')$. An infinite process S may-converges (denoted as $S{\downarrow_{CHFI}}$) if there exists a successful process S' such that $S \xrightarrow{CHFI,*} S'$. Process S should-converges if any successor w.r.t. $\xrightarrow{CHFI}$ may-converges, i.e. $S{\Downarrow_{CHFI}}$ iff $\forall S' : S \xrightarrow{CHFI,*} S' \implies S'{\downarrow_{CHFI}}$. We write $S{\Uparrow_{CHFI}}$ if $S{\downarrow_{CHFI}}$ does not hold (S must-diverges), and we write $S{\uparrow_{CHFI}}$ if $S{\Downarrow_{CHFI}}$ does not hold (S may-diverges).

We will now show that $s \leq^o_{b,PFMI} t$ implies $s \leq_{c,CHFI} t$. More technically, we show that $s \leq^o_{b,PFMI} t$ implies $\mathbb{C}[s]{\downarrow_{CHFI}} \implies C[t]{\downarrow_{CHFI}}$ and $\mathbb{C}[s]{\uparrow_{CHFI}} \implies C[t]{\uparrow_{CHFI}}$ for all infinite process contexts $\mathbb{C}$ with an expression hole.

In the following, we drop the distinction between MVar-constants and variables. Note that this change does not make a difference in convergence behavior.

Let *GCtxt* be process-contexts with several holes, where the holes appear only in subcontexts $x \Leftarrow [\cdot]$ or $x\,\mathbf{m}\,[\cdot]$. We assume that $\mathbb{G} \in$ *GCtxt* is in prenex normal form (i.e. all ν-binders are on the top), that we can rearrange the concurrent processes as in a multiset exploiting that the parallel composition is associative and commutative, and we write $\nu\mathcal{X}.\mathbb{G}'$ where $\nu\mathcal{X}$ represents the whole ν-prefix.

Lemma 5.1. *If $s_i \leq^o_{b,PFMI} t_i$ for $i = 1,\ldots,n$ implies $\mathbb{G}[s_1,\ldots,s_n]{\downarrow_{CHFI}} \implies \mathbb{G}[t_1,\ldots,t_n]{\downarrow_{CHFI}}$ and $\mathbb{G}[t_1,\ldots,t_n]{\uparrow_{CHFI}} \implies \mathbb{G}[s_1,\ldots,s_n]{\uparrow_{CHFI}}$ for all $\mathbb{G} \in$ GCtxt, then $s \leq^o_{b,PFMI} t$ implies $s \leq_{c,CHFI} t$.*

Proof. Let $s \leq^o_{b,PFMI} t$ and $\mathbb{C}[\cdot]$ be a process context with expression hole. Let $\mathbb{C} = \mathbb{C}_1[\mathbb{C}_2]$, such that $\mathbb{C}_2$ is a maximal expression context. Then $\mathbb{C}_2[s] \leq^o_{b,PFMI} \mathbb{C}_2[t]$, since $\leq^o_{b,PFMI}$ is a precongruence. The precondition on the *GCtxt*-contexts now shows that $\mathbb{C}_1[\mathbb{C}_2[s]]{\downarrow_{CHFI}} \implies \mathbb{C}_1[\mathbb{C}_2[t]]{\downarrow_{CHFI}}$ and $\mathbb{C}_1[\mathbb{C}_2[t]]{\uparrow_{CHFI}} \implies \mathbb{C}_1[\mathbb{C}_2[s]]{\uparrow_{CHFI}}$, hence $s \leq_{c,CHFI} t$.

Proposition 5.2. *Let s_i, t_i be pure expressions with $s_i \leq^o_{b,PFMI} t_i$, and let $\mathbb{G} \in$ GCtxt. Then $\mathbb{G}[s_1,\ldots,s_n]{\downarrow_{CHFI}} \implies \mathbb{G}[t_1,\ldots,t_n]{\downarrow_{CHFI}}$.*

Proof. Let $\mathbb{G}[s_1,\ldots,s_n]{\downarrow_{CHFI}}$. We use induction on the number of reductions of $\mathbb{G}[s_1,\ldots,s_n]$ to a successful process. In the base case $\mathbb{G}[s_1,\ldots,s_n]$ is successful. Then either $\mathbb{G}[t_1,\ldots,t_n]$ is also successful, or $\mathbb{G} = \nu\mathcal{X}.x \overset{\mathsf{main}}{\Longleftarrow} [\cdot] \mid \mathbb{G}'$, and w.l.o.g. this is the hole with index 1, and $s_1 = \texttt{return}\ s'_1$. Since $s_1 \leq^o_{b,PFMI} t_1$, there is a reduction $t_1 \xrightarrow{PFMI,*} \texttt{return}\ t'_1$. This reduction is also a *CHFI*-standard reduction of $\mathbb{G}[t_1,\ldots,t_n]$ to a successful process.

Now let $\mathbb{G}[s_1,\ldots,s_n] \xrightarrow{CHFI} S_1$ be the first step of a reduction to a successful process. We analyze the different reduction possibilities:

If the reduction is within some s_i, i.e. $s_i \to s'_i$ by (beta), (case) or (seq), then we can use induction, since the standard-reduction is deterministic within the expression, and a standard reduction of $\mathbb{G}[s_1,\ldots,s_n]$; and since $s_i \sim^o_{b,PFMI} s'_i$.

If the reduction is (lunit), i.e. $\mathbb{G} = \nu\mathcal{X}.x \Leftarrow [\cdot] \mid \mathbb{G}'$, where $s_1 = \mathbb{M}_1[\texttt{return}\ r_1 \texttt{ >>= } r_2]$, and the reduction result of $\mathbb{G}[s_1,\ldots,s_n]$ is $\mathbb{G} = \nu\mathcal{X}.x \Leftarrow \mathbb{M}_1[r_2\ r_1] \mid \mathbb{G}'[s_2,\ldots,s_n]$. We have $s_1 \leq^o_{b,PFMI} t_1$. Let $\mathbb{M}_1 = \mathbb{M}_{1,1}\ldots\mathbb{M}_{1,k}$, where $\mathbb{M}_{1,j} = [\cdot] \texttt{ >>= } s'_j$. By induction on the depth, there is a reduction sequence $t_1 \xrightarrow{CHFI,*} \mathbb{M}_{2,1}\ldots\mathbb{M}_{2,k}[(t'_1 \texttt{ >>= } t'_2)]$, where $\mathbb{M}_{2,j} = [\cdot] \texttt{ >>= } r'_j$, $s'_j \leq^o_{b,PFMI} r'_j$, and $\texttt{return}\ r_1 \leq^o_{b,PFMI} t'_1$. Let $\mathbb{M}_2 := \mathbb{M}_{2,1}\ldots\mathbb{M}_{2,k}$. This implies $t'_1 \xrightarrow{CHFI} \texttt{return}\ t''_1$ with $r_1 \leq^o_{b,PFMI} t''_1$. This reduction is also a standard reduction of the whole process. The corresponding results are $r_2\ r_1$ and $t'_2\ t''_1$. Thus there is a reduction sequence $\mathbb{G}[t_1,\ldots,t_n] \xrightarrow{CHFI,*} \nu\mathcal{X}.x \Leftarrow \mathbb{M}_2[t'_2\ t''_1] \mid \mathbb{G}'[s_2,\ldots,s_n]$. Since $\leq^o_{b,PFMI}$ is a precongruence we have that $\mathbb{M}_1[r_2\ r_1] \leq^o_{b,PFMI} \mathbb{M}_2[t'_2\ t''_1]$ satisfy the induction hypothesis.

For the reductions (tmvar), (pmvar), (nmvar), or (fork) the same arguments as for (lunit) show that the first reduction steps permit to apply the induction hypothesis with the following differences: For the reductions (tmvar) and (pmvar) Proposition 4.11 is used to show that the reduction of $\mathbb{G}[t_1,\ldots,t_n]$ also leads to an MVar-variable in the case $x \leq^o_{b,PFMI} t$. Also the $\mathbb{G}$-hole is transported between the thread and the data-component of the MVar. In case of (fork), the number of holes of the successor $\mathbb{G}'$ of $\mathbb{G}$ may have one more hole.

For (unIOTr) as argued above, $\mathbb{G}[t_1,\ldots,t_n]$ can be reduced such that also a (unIOTr) reduction for $\mathbb{G}[t_1,\ldots,t_n]$ is possible. Assume that the substitutions are $\sigma_s = s' \mathbin{/\!/} x$ and $\sigma_t = t' \mathbin{/\!/} x$ for $\mathbb{G}[s_1,\ldots,s_n]$ and the reduction-successor of $\mathbb{G}[t_1,\ldots,t_n]$. Lemma 4.6 shows that $\sigma_s(s'') \leq^o_{b,PFMI} \sigma_t(t'')$ whenever $s'' \leq^o_{b,PFMI} t''$, and thus the induction hypothesis can be applied. In this step, the number of holes of $\mathbb{G}$ may increase, such that also expression components of MVars may be holes, since the replaced variable x may occur in several places. □

Example 5.3. Let $\mathbb{G}[\cdot] := z \overset{\mathsf{main}}{\Longleftarrow} \texttt{takeMVar}\ x \mid y \Leftarrow [\cdot] \mid x\,\mathbf{m}\,e$, and let $s := \texttt{Bot}$, $t := \texttt{takeMVar}\ x$. Then $s \leq^o_{b,PFMI} t$, $\mathbb{G}[s]{\Downarrow_{CHFI}}$, but $\mathbb{G}[t]{\uparrow_{CHFI}}$. Hence $s \leq^o_{b,PFMI} t$ and $\mathbb{G}[s]{\Downarrow_{CHFI}}$ do not imply $\mathbb{G}[t]{\Downarrow_{CHFI}}$.

Proposition 5.4. *Let s_i, t_i be PFMI-expressions with $s_i \sim^o_{b,PFMI} t_i$, and let $\mathbb{G} \in GCtxt$. Then $\mathbb{G}[s_1,\ldots,s_n]{\Downarrow_{CHFI}} \implies \mathbb{G}[t_1,\ldots,t_n]{\Downarrow_{CHFI}}$.*

Proof. We prove the converse implication: $\mathbb{G}[t_1,\ldots,t_n]{\uparrow_{CHFI}} \implies \mathbb{G}[s_1,\ldots,s_n]{\uparrow_{CHFI}}$. Let $\mathbb{G}[t_1,\ldots,t_n]{\uparrow_{CHFI}}$. We use induction on the number of reductions of $\mathbb{G}[t_1,\ldots,t_n]$ to a must-divergent process. In the base case $\mathbb{G}[t_1,\ldots,t_n]{\Uparrow_{CHFI}}$. Proposition 5.2 shows $\mathbb{G}[s_1,\ldots,s_n]{\Uparrow_{CHFI}}$.

Now let $\mathbb{G}[t_1,\ldots,t_n] \xrightarrow{CHFI} S_1$ be the first reduction of a reduction sequence R to a must-divergent process. We analyze the different reduction possibilities:

If the reduction is within some t_i, i.e. $t_i \to t_i'$ and hence $t_i \sim^o_{b,PFMI} t_i'$, then we use induction, since the reduction is a standard-reduction of $\mathbb{G}[t_1,\ldots,t_n]$.

Now assume that the first reduction step of R is (lunit). I.e., $\mathbb{G} = \nu\mathcal{X}.x \Leftarrow [\cdot] \mid \mathbb{G}'$, where $t_1 = \mathbb{M}[\texttt{return}\ r_1 \texttt{ >>= } r_2]$, and the reduction result of $\mathbb{G}[t_1,\ldots,t_n]$ is $\mathbb{G} = \nu\mathcal{X}.x \Leftarrow \mathbb{M}[r_2\ r_1] \mid \mathbb{G}'[t_2,\ldots,t_n]$. We have $s_1 \sim^o_{b,PFMI} t_1$.

By induction on the reductions and the length of the path to the hole of $\mathbb{M}[\cdot]$, we see that $s_1 \xrightarrow{*} \mathbb{M}_1[\texttt{return}\ r_1' \texttt{ >>= } r_2']$. Then we can perform the (lunit)-reduction and obtain $\mathbb{M}_1[r_2'\ r_1']$. Since $r_2'\ r_1' \sim^o_{b,PFMI} r_2\ r_1$, we obtain a reduction result that satisfies the induction hypothesis.

The other reductions can be proved similarly, using techniques as in the previous case and the proof of Proposition 5.2. For (unIOTr), Lemma 4.6 shows that for the substitutions $\sigma := s \mathbin{/\!/} x$ and $\sigma' := s' \mathbin{/\!/} x$ with $s \sim^o_{b,PFMI} s'$, we have $\sigma(r) \sim^o_{b,PFMI} \sigma(r')$ for expressions r, r' with $r \sim^o_{b,PFMI} r'$, hence the induction can also be used in this case. □

Propositions 5.2 and 5.4 and Lemma 5.1 imply:

Theorem 5.5. *Let $s, t \in IExpr_{PFMI}$ with $s \sim^o_{b,PFMI} t$. Then $s \sim_{c,CHFI} t$.*

6 Conservativity of *PF* in *CHF*

In this section we will prove that contextual equality in *PF* implies contextual equality in *CHF*, i.e. *CHF* is a conservative extension of *PF* w.r.t. contextual equivalence. In a second part we show that this result does not hold, if we add so-called lazy futures to *CHF*. We will now use a translation from [24] which translates *CHF*-processes into *CHFI*-process by removing `letrec`- and shared bindings. This will enable us to show that contextual equality in the pure calculus *PF* implies contextual equality in *CHF*.

Definition 6.1 ([24]). *Let P be a process. The translation $IT :: Proc \to IProc$ translates a process P into its infinite tree process $IT(P)$. It recursively unfolds all bindings of* `letrec`*- and top-level bindings where cyclic variable chains $x_1 = x_2, \ldots, x_n = x_1$ are removed and all occurrences of x_i on other positions are replaced by the new constant* `Bot`*. Top-level bindings are replaced by a* **0***-component. Free variables, futures, and names of MVars are kept in the tree (are not replaced). Equivalence of infinite processes is syntactic, where α-equal trees are assumed to be equivalent. Similarly, IT is also defined for expressions to translate PFI-expressions into PF-expressions.*

Theorem 6.2 ([24]). *For all processes $P \in Proc_{CHF}$ it holds: $P{\downarrow_{CHF}} \iff IT(P){\downarrow_{CHFI}}$ and $P{\Downarrow_{CHF}} \iff IT(P){\Downarrow_{CHFI}}$.*

We first consider *PF*- and *PFI*-expressions:

Proposition 6.3. *Let e_1, e_1 be PF-expressions. Then $e_1 \leq_{c,PF} e_2$ iff $IT(e_1) \leq_{c,PFI} IT(e_2)$.*

Proof. From Theorem 6.2 it easily follows that $IT(e_1) \leq_{c,PFI} IT(e_2)$ implies $e_1 \leq_{c,PF} e_2$. For the other direction, we have to note that there are infinite expressions that are not $IT()$-images of *PF*-expressions. We give a sketch of the proof: Let e_1, e_2 be *PF*-expressions with $e_1 \leq_{c,PF} e_2$. Let $\mathbb{C}$ be a *PFI*-context such that $\mathbb{C}[IT(e_1)]{\downarrow_{PFI}}$. We have to show that also $\mathbb{C}[IT(e_2)]{\downarrow_{PFI}}$. Since $\mathbb{C}[IT(e_1)]{\downarrow_{PFI}}$ by a finite reduction, there is a finite context $\mathbb{C}'$ such that $\mathbb{C}'$ can be derived from $\mathbb{C}$ by replacing subexpressions by `Bot`, with $\mathbb{C}'[IT(e_1)]{\downarrow_{PFI}}$. Since equivalence of convergence holds and since $\mathbb{C}'$ is invariant under IT, this shows $\mathbb{C}'[e_1]{\downarrow_{PF}}$. The assumption shows $\mathbb{C}'[e_2]{\downarrow_{PF}}$. This implies $\mathbb{C}'[IT(e_2)]{\downarrow_{PFI}}$. Standard reasoning shows that also $\mathbb{C}[IT(e_2)]{\downarrow_{PFI}}$.

Main Theorem 6.4 *Let $e_1, e_2 \in Expr_{PF}$. Then $e_1 \sim_{c,PF} e_2$ iff $e_1 \sim_{c,CHF} e_2$.*

Proof. One direction is trivial. For the other direction the reasoning is as follows: Let e_1, e_2 be *PF*-expressions. Then Proposition 6.3 shows that $e_1 \sim_{c,PF} e_2$ is equivalent to $IT(e_1) \sim_{c,PFI} IT(e_2)$. Now Theorem 4.4 and Proposition 4.10 show that $IT(e_1) \sim_{b,PFMI} IT(e_2)$. Then Theorem 5.5 shows that $IT(e_1) \sim_{c,CHFI} IT(e_2)$. Finally, from Theorem 6.2 it easily follows that $e_1 \sim_{c,CHF} e_2$. □

6.1 Lazy Futures Break Conservativity

Having proved our main result, we now show that there are innocent looking extensions of CHF that break the conservativity result. One of those are so-called lazy futures. The equivalence $\texttt{seq}\, e_1\, e_2$ and $\texttt{seq}\, e_2\, (\texttt{seq}\, e_1\, e_2)$ used by Kiselyov's counter example [9], holds in the pure calculus and in *CHF* (see Appendix B). This implies that Kiselyov's counter example cannot be transferred to CHF.

Let the calculus *CHFL* be an extension of *CHF* by a lazy future construct, which implements the idea of implementing futures that can be generated as non-evaluating, and which have to be activated by an (implicit) call from another future. We show that this construct would destroy conservativity.

We add a process component $x \overset{lazy}{\Longleftarrow} e$ which is a *lazy future*, i.e. a thread which can not be reduced unless its evaluation is forced by another thread. On the expression level we add a construct `lfuture` of type $\texttt{IO}\ \tau \rightarrow \texttt{IO}\ \tau$. The operational semantics is extended by two additional reduction rules:

$$\begin{array}{ll}
\text{(lfork)} & y \Leftarrow \mathbb{M}[\texttt{lfuture}\ e] \rightarrow y \Leftarrow \mathbb{M}[\texttt{return}\ x] \mid x \overset{lazy}{\Longleftarrow} e \\
\text{(force)} & y \Leftarrow \mathbb{M}[\mathbb{F}[x]] \mid x \overset{lazy}{\Longleftarrow} e \rightarrow y \Leftarrow \mathbb{M}[\mathbb{F}[x]] \mid x \Leftarrow e
\end{array}$$

The rule (lfork) creates a lazy future. Evaluation can turn a lazy future into a concurrent future if its value is demanded (rule (force)).

In CHF the equation $(\texttt{seq}\, e_2\, (\texttt{seq}\, e_1\, e_2)) \sim_{Bool} (\texttt{seq}\, e_1\, e_2)$ for $e_1, e_2 :: Bool$ holds (see above) The equation does not hold in *CHFL*. Consider the context that uses lazy futures and distinguishes the two expressions:

$$\begin{array}{l}
\mathbb{C} = z \overset{\textsf{main}}{\Longleftarrow} \texttt{case}_{\text{Bool}}\ [\cdot]\ \texttt{of}\ (\texttt{True} \rightarrow \bot)\ (\texttt{False} \rightarrow \texttt{return True}) \mid v\, \mathbf{m}\, \texttt{True} \\
\quad \mid x \overset{lazy}{\Longleftarrow} \texttt{takeMVar}\ v \texttt{>>=} \lambda w.(\texttt{putMVar}\ v\ \texttt{False} \texttt{>>=} \lambda\_ \rightarrow \texttt{return}\ w) \\
\quad \mid y \overset{lazy}{\Longleftarrow} \texttt{takeMVar}\ v \texttt{>>=} \lambda w.(\texttt{putMVar}\ v\ \texttt{False} \texttt{>>=} \lambda\_ \rightarrow \texttt{return}\ w)
\end{array}$$

Then $\mathbb{C}[\texttt{seq}\ y\ (\texttt{seq}\ x\ y)]$ must-diverges, since its evaluation (deterministically) results in $z \overset{\textsf{main}}{\Longleftarrow} \bot \mid x = \texttt{False} \mid y = \texttt{True} \mid v\, \mathbf{m}\, \texttt{False}$. On the other hand $C[\texttt{seq}\ x\ y]\Downarrow_{CHFL}$, since it evaluates to $z \overset{\textsf{main}}{\Longleftarrow} \texttt{True} \mid x \Leftarrow \texttt{True} \mid y \Leftarrow \texttt{False} \mid v\, \mathbf{m}\, \texttt{False}$ where again the evaluation is deterministic. Thus context $\mathbb{C}$ distinguishes $\texttt{seq}\ x\ y$ and $\texttt{seq}\ y\ (\texttt{seq}\ x\ y)$ w.r.t. $\sim_c$.

Hence adding an `unsafeInterleaveIO`-operator to *CHF* results in the loss of conservativity, since lazy futures can be implemented in CHF (or even in Concurrent Haskell) using `unsafeInterleaveIO` to delay the thread creation:

```
lfuture act = unsafeInterleaveIO (do ack ← newEmptyMVar
                                     thread ← forkIO (act >>= putMVar ack)
                                     takeMVar ack)
```

7 Conclusion

We have shown that the calculus *CHF* modelling most features of Concurrent Haskell with `unsafeInterleaveIO` is a conservative extension of the pure language, and exhibited a counterexample showing that adding the unrestricted use of `unsafeInterleaveIO` is not. This complements our results in [24] where correctness of monad laws was shown, provided that the type of the first argument of `seq` is restricted to functional types. Future work is to also analyze further extensions like killing threads, and synchronous and asynchronous exceptions (as in [11, 17]), where our working hypothesis is that killing threads and (at least) synchronous exceptions retain the our conservativity result.

References

1. S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pp. 65–116. Addison-Wesley, 1990.
2. H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. In *Symp. on Artificial intelligence and programming languages 1977*, pp. 55–59, 1977. ACM.
3. A. Carayol, D. Hirschkoff, and D. Sangiorgi. On the representation of McCarthy's amb in the pi-calculus. *Theoret. Comput. Sci.*, 330(3):439–473, 2005.
4. R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoret. Comput. Sci.*, 34:83–133, 1984.
5. R. H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, 1985.
6. D. Howe. Equality in lazy computation systems. In *4th LICS*, pp. 198–203, 1989.
7. D. Howe. Proving congruence of bisimulation in functional programming languages. *Inform. and Comput.*, 124(2):103–112, 1996.
8. P. Johann and J. Voigtländer. The impact of seq on free theorems-based program transformations. *Fund. Inform.*, 69(1–2):63–102, 2006.
9. O. Kiselyov. Lazy IO breaks purity, 2009. Haskell Mailinglist, 4. March 2009, http://www.haskell.org/pipermail/haskell/2009-March/021064.html.
10. M. Mann and M. Schmidt-Schauß. Similarity implies equivalence in a class of non-deterministic call-by-need lambda calculi. *Inform. and Comput.*, 208(3):276 – 291, 2010.
11. S. Marlow, S. L. P. Jones, A. Moran, and J. H. Reppy. Asynchronous exceptions in Haskell. In *PLDI*'01, pp. 274–285, 2001.
12. R. Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge university press, 1999. ISBN 0-521-65869-1.
13. K. Nakata and M. Hasegawa. Small-step and big-step semantics for call-by-need. *J. Funct. Program.*, 19:699–722, 2009.
14. J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoret. Comput. Sci.*, 364(3):338–356, 2006.
15. J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *Electron. Notes Theor. Comput. Sci.*, 173:313–337, 2007.
16. S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report.* Cambridge University Press, 2003. `www.haskell.org`.
17. S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pp. 47–96. IOS-Press, 2001.
18. S. Peyton Jones and S. Singh. A tutorial on parallel and concurrent programming in Haskell. In *AFP'08*, pp. 267–305, 2009. Springer.

19. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL' 96*, pp. 295–308. ACM, 1996.
20. S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *POPL'93*, pp. 71–84. ACM, 1993.
21. A. M. Pitts. Howe's method for higher-order languages. In *Advanced Topics in Bisimulation and Coinduction*, *Cambridge Tracts in Theoretical Computer Science* 52, chapter 5, pp. 197–232. Cambridge University Press, to appear Nov. 2011.
22. A. Rensink and W. Vogler. Fair testing. *Inform. and Comput.*, 205(2):125–198, 2007.
23. D. Sabel and M. 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.
24. D. Sabel and M. Schmidt-Schauß. A contextual semantics for concurrent Haskell with futures. In *PPDP'11*, pp. 101–112, 2011. ACM.
25. D. Sangiorgi and D. Walker. *The π-calculus: a theory of mobile processes*. Cambridge university press, 2001.
26. M. Schmidt-Schauß and D. Sabel. Closures of may-, should- and must-convergences for contextual equivalence. *Inform. Process. Lett.*, 110(6):232 – 235, 2010.
27. M. Schmidt-Schauß, M. Schütz, and D. Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.
28. P. Sestoft. Deriving a lazy abstract machine. *J. Funct. Programming*, 7(3):231–264, 1997.
29. H. Søndergaard and P. Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27:505–517, 1989.
30. P. Wadler. Monads for functional programming. In *AFP'95*, *LNCS* 925, pp. 24–52. Springer, 1995.

A The Congruence Proof

The goal of this section is to show that $\leq_{b,PFMI}$ and $\leq_{b,PFI}$ are precongruences. We omit the proof for the calculus *PFI* and only consider *PFMI*, since the proofs for *PFI* are completely analogous. The proof method used below for showing that similarity is a precongruence is derived from Howe [6], though extended to infinite expressions. For a developed proof for may-convergence in a non-deterministic setting with finite expressions, see [10].
The fixpoint property of $\leq_{b,PFMI}$ implies:

Lemma A.1. *For closed values $\theta(s_1 \dots s_n), \theta(t_1 \dots t_n)$, we have $\theta(s_1 \dots s_n) \leq_{b,PFMI} \theta(t_1 \dots t_n)$ iff $s_i \leq^o_{b,PFMI} t_i$.*
In the concrete syntax, if θ is a constructor or a monadic operator, then $\theta(s_1 \dots s_n) \leq_{b,PFMI} \theta(t_1 \dots t_n)$ iff $s_i \leq_{b,PFMI} t_i$, and $\lambda x.s \leq_{b,PFMI} \lambda x.t$ iff $s \leq^o_{b,PFMI} t$.

Lemma A.2. *The relations $\leq_{b,PFMI}$ and $\leq^o_{b,PFMI}$ are reflexive and transitive.*

Proof. Reflexivity is obvious. Transitivity follows by showing that $\eta := \leq_{b,PFMI} \cup (\leq_{b,PFMI} \circ \leq_{b,PFMI})$ satisfies $\eta \subseteq F_{PFMI}(\eta)$ and then using the coinduction principle.

The goal in the following is to show that $\leq_{b,PFMI}$ is a precongruence. A relation μ is *operator-respecting*, iff $s_i \;\mu\; t_i$ for $i = 1, \dots, n$ implies $\xi(s_1, \dots, s_n) \;\mu\; \xi(t_1, \dots, t_n)$. This proof proceeds by defining a congruence candidate $\leq_{cand}$ as a closure of $\leq_{b,PFMI}$ within contexts, which obviously is operator respecting: This relation is not known to be transitive. Then we show that $\leq_{b,PFMI}$ and $\leq_{cand}$ coincide.

Definition A.3. *The precongruence candidate $\leq_{cand}$ is a binary relation on open expressions and is defined as the greatest fixpoint of the operator F_{cand} on relations on all expressions:*

1. *$x \; F_{cand}(\eta) \; s$ iff $x \leq^o_{b,PFMI} s$.*
2. *$\xi(s_1, \dots, s_n) \; F_{cand}(\eta) \; s$ iff there is some expression $\xi(s'_1, \dots, s'_n) \leq^o_{b,PFMI} s$ with $s_i \;\eta\; s'_i$ for $i = 1, \dots, n$.*

The operator F_{cand} is monotone, hence the definition makes sense. Presumably it is not continuous, hence usual induction over an $\mathbb{N}$-indexed intersection does not work and we have to stick to coinduction for the proofs:

Lemma A.4. *If some relation η satisfies $\eta \subseteq F_{cand}(\eta)$, then $\eta \subseteq \; \leq_{cand}$.*

Since $\leq_{cand}$ is a fixpoint of F_{cand}, we have:

Lemma A.5.

1. *$x \leq_{cand} s$ iff $x \leq^o_{b,PFMI} s$.*
2. *$\xi(s_1, \dots, s_n) \leq_{cand} s$ iff there is some expression $\xi(s'_1, \dots, s'_n) \leq^o_{b,PFMI} s$ with $s_i \leq_{cand} s'_i$ for $i = 1, \dots, n$.*

Some technical facts about the precongruence candidate are now proved:

Lemma A.6.

1. $\leq_{cand}$ *is reflexive.*
2. $\leq_{cand}$ *and* $(\leq_{cand})^c$ *are operator-respecting.*
3. $\leq^o_{b,PFMI} \subseteq \leq_{cand}$ *and* $\leq_{b,PFMI} \subseteq (\leq_{cand})^c$.
4. $\leq_{cand} \circ \leq^o_{b,PFMI} \subseteq \leq_{cand}$.
5. $(s \leq_{cand} s' \wedge t \leq_{cand} t') \implies t[s/x] \leq_{cand} t'[s'/x]$.
6. $s \leq_{cand} t$ *implies that* $\sigma(s) \leq_{cand} \sigma(t)$ *for every substitution* σ.
7. $\leq_{cand} \subseteq ((\leq_{cand})^c)^o$

Proof. 1. This follows from Lemma A.5, since $\leq^o_b$ is reflexive, using coinduction: Show that $\eta := \leq_{cand} \cup \{(s,s) \mid s \in \mathit{IExpr}_{PFMI}\}$ satisfies $\eta \subseteq F_{cand}(\eta)$.
2. Let η be the operator-respecting closure of $\leq_{cand}$. I.e., the least fixpoint of adding relations $\xi(s_1,\ldots,s_n)\ \eta\ \xi(t_1,\ldots,t_n)$ if $s_i\ \eta\ t_i$ for all i, starting with $\leq_{cand}$. We will show that $\eta \subseteq F_{cand}(\eta)$. So assume that $\xi(s_1,\ldots,s_n)\ \eta\ \xi(t_1,\ldots,t_n)$ holds. If $\xi(s_1,\ldots,s_n) \leq_{cand} \xi(t_1,\ldots,t_n)$, then $\xi(s_1,\ldots,s_n)\ F_{cand}(\eta)\ \xi(t_1,\ldots,t_n)$, since $\leq_{cand} \subseteq \eta$, and $\leq_{cand}$ is the greatest fixpoint of F_{cand}. Otherwise $\xi(s_1,\ldots,s_n)\ \eta\ \xi(t_1,\ldots,t_n)$ since $s_i\ \eta\ t_i$ for all i. Then $\xi(s_1,\ldots,s_n)\ F_{cand}(\eta)\ \xi(t_1,\ldots,t_n)$ since $\leq^o_{b,PFMI}$ is reflexive. By coinduction we obtain $\eta \subseteq \leq_{cand}$. Since also $\leq_{cand} \subseteq \eta$, we have $\eta = \leq_{cand}$.
3. This follows from Lemma A.5, since $\leq_{cand}$ is reflexive.
4. This follows from the definition, Lemma A.5 and transitivity of $\leq^o_{b,PFMI}$.
5. Let $\eta := \leq_{cand} \cup \{(r[s/x], r'[s'/x]) \mid r \leq_{cand} r'\}$. We show that $\eta \subseteq F_{cand}(\eta)$: In the case $x \leq_{cand} r'$, we obtain $x \leq^o_{b,PFMI} r'$ from the definition, and $s' \leq^o_{b,PFMI} r'[s'/x]$ and thus $x[s/x] \leq_{cand} r'[s'/x]$. In the case $y \leq_{cand} r$, we obtain $y \leq^o_{b,PFMI} r'$ from the definition, and $y[s/x] = y \leq^o_{b,PFMI} r'[s'/x]$ and thus $y = y[s/x] \leq_{cand} r'[s'/x]$. If $r = \xi(r_1,\ldots,r_n)$ and $r \leq_{cand} r'$ and $r[s/x]\ \eta\ r'[s'/x]$. Then there is some $\xi(r'_1,\ldots,r'_n) \leq^o_{b,PFMI} r'$ with $r_i \leq_{cand} r'_i$. W.l.o.g. bound variables have fresh names. We have $r_i[s/x]\ \eta\ r'_i[s'/x]$ and $\xi(r'_1,\ldots,r'_n)[s'/x] \leq^o_{b,PFMI} r'[s'/x]$. Thus $r[s/x]\ F_{cand}(\eta)\ r'[s'/x]$. By coinduction we see that $\leq_{cand} = \eta$.
6. This follows from item 5.
7. This follows from item 6 and Lemma 3.4. □

Lemma A.7. *The middle expression in the definition of* $\leq_{cand}$ *can be chosen as closed, if* s, t *are closed: Let* $s = \xi(s_1,\ldots,s_{ar(\xi)})$*, such that* $s \leq_{cand} t$ *holds. Then there are operands* s'_i*, such that* $\xi(s'_1,\ldots,s'_{ar(\xi)})$ *is closed,* $\forall i : s_i \leq_{cand} s'_i$ *and* $\xi(s'_1,\ldots,s'_{ar(\xi)}) \leq^o_{b,PFMI} s$.

Proof. The definition of $\leq_{cand}$ implies that there is an expression $\xi(s''_1,\ldots,s''_{ar(\xi)})$ such that $s_i \leq_{cand} s''_i$ for all i and $\xi(s''_1,\ldots,s''_{ar(\xi)}) \leq^o_{b,PFMI} t$. Let σ be the substitution with $\sigma(x) := v_x$ for all $x \in FV(\xi(s''_1,\ldots,s''_{ar(\xi)}))$, where

v_x is any closed expression. Note that for every type τ there exists a closed expression, namely `Bot` $:: \tau$. Lemma A.6 now shows that $s_i = \sigma(s_i) \leq_{cand} \sigma(s''_i)$ holds for all i. The relation $\sigma(\xi(s''_1, \ldots, s''_{ar(\xi)})) \leq^o_{b,PFMI} t$ holds, since t is closed and due to the definition of an open extension. The requested expression is $\xi(\sigma(s''_1), \ldots, \sigma(s''_{ar(\xi)}))$.

Lemmas 3.7 and A.6 imply that $\leq_{cand}$ is right-stable w.r.t. reduction:

Lemma A.8. *If $s \leq_{cand} t$ and $t \xrightarrow{PFMI} t'$, then $s \leq_{cand} t'$.*

We show that $\leq_{cand}$ is left-stable w.r.t. reduction:

Lemma A.9. *Let s, t be closed expressions such that $s = \theta(s_1, \ldots, s_n)$ is a value and $s \leq_{cand} t$. Then there is some closed value $t' = \theta(t_1, \ldots, t_n)$ with $t \xrightarrow{PFMI,*} t'$ and for all $i : s_i \leq_{cand} t_i$.*

Proof. The definition of $\leq_{cand}$ implies that there is a closed expression $\theta(t'_1, \ldots, t'_n)$ with $s_i \leq_{cand} t'_i$ for all i and $\theta(t'_1, \ldots, t'_n) \leq_{b,PFMI} t$. Consider the case $s = \lambda x.s'$. Then there is some closed $\lambda x.t' \leq_{b,PFMI} t$ with $s' \leq_{cand} t'$. The relation $\lambda x.t' \leq_{b,PFMI} t$ implies that $t \xrightarrow{PFMI,*} \lambda x.t''$. Lemma 3.7 now implies $\lambda x.s' \leq_{cand} \lambda x.t''$. Definition of $\leq_{cand}$ and Lemma A.7 now show that there is some closed $\lambda x.t^{(3)}$ with $s' \leq_{cand} t^{(3)}$ and $\lambda x.t^{(3)} \leq_{b,PFMI} \lambda x.t''$. The latter relation implies $t^{(3)} \leq^o_{b,PFMI} t''$, which shows $s' \leq_{cand} t''$ by Lemma A.6 (4).

If θ is a constructor, then there is a closed expression $\theta(t'_1, \ldots, t'_n)$ with $s_i \leq_{cand} t'_i$ for all i and $\theta(t'_1, \ldots, t'_n) \leq_{b,PFMI} t$. The definition of $\leq_{b,PFMI}$ implies that $t \xrightarrow{PFMI,*} \theta(t''_1, \ldots, t''_n)$ with $t'_i \leq_{b,PFMI} t''_i$ for all i. By definition of $\leq_{cand}$, we obtain $s_i \leq_{cand} t''_i$ for all i.

Proposition A.10. *Let s, t be closed expressions, $s \leq_{cand} t$ and $s \xrightarrow{PFMI} s'$ where s is the redex. Then $s' \leq_{cand} t$.*

Proof. The relation $s \leq_{cand} t$ implies that $s = \xi(s_1, \ldots, s_n)$ and that there is some closed $t' = \xi(t'_1, \ldots, t'_n)$ with $s_i \leq_{cand} t'_i$ for all i and $t' \leq^o_{b,PFMI} t$.

- For the (beta)-reduction, $s = s_1\ s_2$, where $s_1 = (\lambda x.s'_1)$, s_2 is a closed term, and $t' = t'_1\ t'_2$. Lemma A.9 and $s_1 \leq_{cand} t'_1$ show that $t'_1 \xrightarrow{PFMI,*} \lambda x.t''_1$ with $\lambda x.s'_1 \leq_{cand} \lambda x.t''_1$ and also $s'_1 \leq_{cand} t''_1$. From $t' \xrightarrow{PFMI,*} t''_1[t'_2/x]$ we obtain $t''_1[t'_2/x] \leq_{b,PFMI} t$. Lemma A.6 now shows $s'_1[s_2/x] \leq_{cand} t''_1[t'_2/x]$. Hence $s'_1[s_2/x] \leq_{cand} t$, again using Lemma A.6.
- Similar arguments apply to the case-reduction.
- Suppose, the reduction is a `seq`-reduction. Then $s \leq_{cand} t$ and $s =$ (`seq` $s_1\ s_2$). Lemma A.7 implies that there is some closed (`seq` $t'_1\ t'_2$) $\leq^o_{b,PFMI}$ t with $s_i \leq_{cand} t'_i$. Since s_1 is a value, Lemma A.9 shows that there is a reduction $t'_1 \xrightarrow{PFMI,*} t''_1$, where t''_1 is a value. There are the reductions $s \xrightarrow{PFMI} s_2$ and (`seq` $t'_1\ t'_2$) $\xrightarrow{PFMI,*}$ (`seq` $t''_1\ t'_2$) $\xrightarrow{PFMI} t'_2$. Since $t'_2 \leq^o_{b,PFMI}$ (`seq` $t'_1\ t'_2$) $\leq^o_{b,PFMI} t$, and $s_2 \leq_{cand} t'_2$, we obtain $s_2 \leq_{cand} t$. $\square$

Proposition A.11. *Let s, t be closed expressions, $s \leq_{cand} t$ and $s \xrightarrow{PFMI} s'$. Then $s' \leq_{cand} t$.*

Proof. We use induction on the length of the path to the hole. The base case is proved in Proposition A.10. Let $\mathbb{E}[s], t$ be closed, $\mathbb{E}[s] \leq_{cand} t$ and $\mathbb{E}[s] \xrightarrow{PFMI} \mathbb{E}[s']$, where we assume that the redex s is not at the top level and that $\mathbb{E}$ is an *IECtxt*-context. The relation $\mathbb{E}[s] \leq_{cand} t$ implies that $\mathbb{E}[s] = \xi(s_1, \ldots, s_n)$ and that there is some closed $t' = \xi(t'_1, \ldots, t'_n) \leq^o_{b,PFMI} t$ with $s_i \leq_{cand} t'_i$ for all i. If $s_j \xrightarrow{PFMI} s'_j$, then by induction hypothesis, $s'_j \leq_{cand} t'_j$. Since $\leq_{cand}$ is operator-respecting, we obtain also $\mathbb{E}[s'] = \xi(s_1, \ldots, s_{j-1}, s'_j, s_{j+1}, \ldots, s_n) \leq_{cand} \xi(t'_1, \ldots, t'_{j-1}, t'_j, t'_{j+1}, \ldots, t'_n)$, and from $\xi(t'_1, \ldots, t'_n) \leq^o_{b,PFMI} t$, also $\mathbb{E}[s'] = \xi(s_1, \ldots, s_{j-1}, s'_j, s_{j+1}, \ldots, s_n) \leq_{cand} t$.

Now we are ready to prove that the precongruence candidate and similarity coincide. First we prove this for the relations on closed expressions and then consider (possibly) open expressions.

Theorem A.12. $(\leq_{cand})^c = \leq_{b,PFMI}$.

Proof. Since $\leq_{b,PFMI} \subseteq (\leq_{cand})^c$ by Lemma A.6, we have to show that $(\leq_{cand})^c \subseteq \leq_{b,PFMI}$. Therefore it is sufficient to show that $(\leq_{cand})^c$ satisfies the fixpoint equation for $\leq_{b,PFMI}$. We show that $(\leq_{cand})^c \subseteq F_{PFMI}((\leq_{cand})^c)$. Let $s\ (\leq_{cand})^c\ t$ for closed terms s, t. We show that $s\ F_{PFMI}((\leq_{cand})^c)\ t$: If $\neg(s{\downarrow_{PFMI}})$, then $s\ F_{PFMI}((\leq_{cand})^c)\ t$ holds by Lemma A.6. If $s{\downarrow_{PFMI}}\theta(s_1, \ldots, s_n)$, then $\theta(s_1, \ldots, s_n)\ (\leq_{cand})^c\ t$ by Lemma A.11. Lemma A.9 shows that $t \xrightarrow{PFMI,*} \theta(t_1, \ldots, t_n)$ and for all $i : s_i \leq_{cand} t_i$. This implies $s\ F_{PFMI}((\leq_{cand})^c)\ t$, since $\theta(t_1, \ldots, t_n) \leq^o_{b,PFMI} t$. We have proved the fixpoint property of $(\leq_{cand})^c$ w.r.t. F_{PFMI}, and hence $(\leq_{cand})^c = \leq_{b,PFMI}$.

Theorem A.13. $\leq_{cand} = \leq^o_{b,PFMI}$.

Proof. Theorem A.12 shows $(\leq_{cand})^c \subseteq \leq_{b,PFMI}$. Hence $((\leq_{cand})^c)^o \subseteq \leq^o_{b,PFMI}$ by monotonicity. Lemma A.6 (7) implies $\leq_{cand} \subseteq ((\leq_{cand})^c)^o \subseteq \leq^o_{b,PFMI}$.

This immediately implies:

Corollary A.14. *$\leq^o_{b,PFMI}$ is a precongruence on infinite expressions $IExpr_{PFMI}$. If σ is a substitution, then $s \leq^o_{b,PFMI} t$ implies $\sigma(s) \leq^o_{b,PFMI} \sigma(t)$.*

The same reasoning can also be performed for $\leq_{b,PFI}$:

Corollary A.15. *$\leq^o_{b,PFI}$ is a precongruence on infinite expressions $IExpr_{PFI}$. If σ is a substitution, then $s \leq^o_{b,PFI} t$ implies $\sigma(s) \leq^o_{b,PFI} \sigma(t)$.*

The last two corollaries show

Theorem A.16. *$\leq^o_{b,PFMI}$ is a precongruence on infinite expressions $IExpr_{PFMI}$. If σ is a substitution, then $s \leq^o_{b,PFMI} t$ implies $\sigma(s) \leq^o_{b,PFMI} \sigma(t)$.*

$\leq^o_{b,PFI}$ is a precongruence on infinite expressions $IExpr_{PFI}$. If σ is a substitution, then $s \leq^o_{b,PFI} t$ implies $\sigma(s) \leq^o_{b,PFI} \sigma(t)$.

A.1 Recursive Replacements

Lemma A.17. *Let x, y be a variables and t_1, t_2 be PFMI-expressions with $x \leq^o_{b,PFMI} t_2$ and $y \leq^o_{b,PFMI} t_1$. Then $x[y \,/\!/\, x] \leq^o_{b,PFMI} t_2[t_1 \,/\!/\, x]$.*

Proof. The relation $y \leq^o_{b,PFMI} t_1$ implies $y \leq^o_{b,PFMI} \sigma(t_1)$ for all substitutions with $\sigma(y) = y$, hence $y \leq^o_{b,PFMI} t_1[t_2 \,/\!/\, x]$.

Lemma A.18. *Let x be a variable and s_1, s_2, t_1, t_2 be PFMI-expressions with $s_i \leq^o_{b,PFMI} t_i$ for $i = 1, 2$. Then $s_2[s_1 \,/\!/\, x] \leq^o_{b,PFMI} t_2[t_1 \,/\!/\, x]$.*

Proof. In the proof we use Theorem A.13 and also the knowledge about $\leq^o_{b,PFMI}$ and F_{cand}. If s_1 is the variable x, then the substitution $[s_1 \,/\!/\, x]$ is $x \mapsto \texttt{Bot}$, and the claim follows easily. Otherwise, we have $s_1 \neq x$. Let ρ be the relation defined by all pairs $s_2[s_1 \,/\!/\, x] \;\rho\; t_2[t_1 \,/\!/\, x]$ for all s_1, s_2, t_1, t_2 with $s_i \leq^o_{b,PFMI} t_i$ for $i = 1, 2$. In order to use coinduction, we show that $\rho \subseteq F_{cand}(\rho)$:
Note that $\leq^o_{b,PFMI} \subseteq \rho$. Assume $s_2[s_1 \,/\!/\, x] \;\rho\; t_2[t_1 \,/\!/\, x]$.

- $s_2[s_1 \,/\!/\, x]$ is a variable. Then it cannot be x. If $s_2 = x$, and $s_1 = y$, then $s_2[s_1 \,/\!/\, x] = y$ and then Lemma A.17 shows $s_2[s_1 \,/\!/\, x] \leq^o_{b,PFMI} t_2[t_1 \,/\!/\, x]$. If $s_2 = y \neq x$, then $s_2[s_1 \,/\!/\, x] = y = s_2[t_1 \,/\!/\, x]$. Since $\leq^o_{b,PFMI}$ is invariant under substitutions, we also obtain $s_2[s_1 \,/\!/\, x] \leq^o_{b,PFMI} t_2[t_1 \,/\!/\, x]$.
- $s_2[s_1 \,/\!/\, x]$ is not a variable. If $s_2 = x$, then $s_2[s_1 \,/\!/\, x] = s_1 \leq^o_{b,PFMI} s_2[t_1 \,/\!/\, x] \leq^o_{b,PFMI} t_2[t_1 \,/\!/\, x]$. If $s_2 = \xi(s'_1, \ldots, s'_n)$, then there is some expression $\xi(t'_1, \ldots, t'_n) \leq^o_{b,PFMI} t_2$ with $s'_i \leq^o_{b,PFMI} t'_i$. Hence $s'_i[s_1 \,/\!/\, x] \;\rho\; t'_i[t_1 \,/\!/\, x]$ by the definition of ρ. This means $s_2[s_1 \,/\!/\, x] \; F_{cand}(\rho) \; t_2[t_1 \,/\!/\, x]$.

Hence coinduction allows us to conclude $\rho \subseteq \leq^o_{b,PFMI}$. Obviously, the other direction also holds, hence $\rho = \leq^o_{b,PFMI}$.

B An Equivalence for seq-Expressions

Before proving Proposition B.2 we show a helpful proposition:

Proposition B.1. *Let s, t be closed infinite PFI-expressions such that $s{\downarrow}v \implies t{\downarrow}v$ where v is a closed value. Then $s \leq_{b,PFI} t$.*

Proof. It easy to verify that the relation $\mathcal{R}_v := \{(s, t) \mid s, t \in \mathit{IExpr}^c, s{\downarrow}v \implies t{\downarrow}v\} \cup \{(s, s) \mid s \in \mathit{IExpr}^c\}$ satisfies $\mathcal{R}_v \subseteq F_{PFI}(\mathcal{R}_v)$. Hence Proposition 3.6 shows $\mathcal{R}_v \subseteq \leq_b$.

Now we prove Proposition B.2. The claim is:

Proposition B.2. *For any (also open) expressions $e_1, e_2 \in \mathit{Expr}_{PF}$ the equality* $\texttt{seq}\ e_1\ e_2 \sim_{c,PF} \texttt{seq}\ e_2\ (\texttt{seq}\ e_1\ e_2)$ *as well as* $\texttt{seq}\ e_1\ e_2 \sim_{c,CHF} \texttt{seq}\ e_2\ (\texttt{seq}\ e_1\ e_2)$ *holds.*

Proof. First we show $\texttt{seq}\ s\ t \leq_{b,PFI} \texttt{seq}\ t\ (\texttt{seq}\ s\ t)$ and $\texttt{seq}\ s\ t \geq_{b,PFI} \texttt{seq}\ t\ (\texttt{seq}\ s\ t)$ for infinite expressions $s, t \in \mathit{Expr}_{PFI}$, where it is sufficient to consider closed terms s, t. If $\texttt{seq}\ s\ t{\downarrow_{PFI}} w$, then clearly there exists a value v such that $\texttt{seq}\ s\ t \xrightarrow{PFI,*} \texttt{seq}\ v\ t \xrightarrow{PFI,seq} t \xrightarrow{PFI,*} w$. Thus we can construct the reduction sequence $\texttt{seq}\ t\ (\texttt{seq}\ s\ t) \xrightarrow{PFI,*} \texttt{seq}\ w\ (\texttt{seq}\ s\ t) \xrightarrow{PFI,seq} \texttt{seq}\ s\ t \xrightarrow{PFI,*} w$.

If $\texttt{seq}\ t\ (\texttt{seq}\ s\ t){\downarrow_{PFI}} w$, then obviously also $\texttt{seq}\ s\ t \xrightarrow{PFI,*} w$. This shows $\texttt{seq}\ s\ t{\downarrow_{PFI}} w$ if, and only if $\texttt{seq}\ t\ (\texttt{seq}\ s\ t){\downarrow_{PFI}} w$. Now Proposition B.1 shows that $\texttt{seq}\ s\ t \sim_{b,PFI} \texttt{seq}\ t\ (\texttt{seq}\ s\ t)$. Proposition 6.3 implies that $\texttt{seq}\ e_1\ e_2 \sim_{c,PF} \texttt{seq}\ e_2\ (\texttt{seq}\ e_1\ e_2)$, which is the first claim. Theorem 5.5 shows $\texttt{seq}\ e_1\ e_2 \sim_{c,CHF} \texttt{seq}\ e_2\ (\texttt{seq}\ e_1\ e_2)$.