Conservative Concurrency in Haskell

David Sabel and Manfred Schmidt-Schauß
Computer Science Institute,
Goethe-University,
Frankfurt am Main, Germany
Email: {sabel,schauss}@ki.informatik.uni-frankfurt.de

***Abstract*—The calculus CHF models Concurrent Haskell extended by concurrent, implicit futures. It is a lambda and process calculus with concurrent threads, monadic concurrent evaluation, and includes a pure functional lambda-calculus PF which comprises data constructors, case-expressions, letrec-expressions, and Haskell's seq. Our main result is conservativity of CHF as extension of PF. This allows us to argue that compiler optimizations and transformations from pure Haskell remain valid in Concurrent Haskell even if it is extended by futures. We also show that conservativity does no longer hold if the extension includes Concurrent Haskell and unsafeInterleaveIO.**

***Keywords*-Contextual equivalence, Concurrency, Functional programming, Semantics, Haskell**

I. Introduction

Pure non-strict functional programming is semantically well understood, permits mathematical reasoning and is referentially transparent [29]. A witness is the core language of the functional part of Haskell [17] consisting only of supercombinator definitions, abstractions, applications, data constructors and case-expressions. However, useful programming languages require much more expressive power for programming and controlling IO-interactions. Haskell employs monadic programming [30], [19] as an interface between imperative and non-strict pure functional programming. However, the sequentialization of IO-operations enforced by Haskell's IO-monad sometimes precludes declarative programming. Thus Haskell implementations provide the primitives `unsafePerformIO` :: `IO a` → `a` which switches off any restrictions enforced by the IO-monad and `unsafeInterleaveIO` :: `IO a` → `IO a` which delays a monadic action inside Haskell's IO-monad. Strict sequentialization is also lost in *Concurrent Haskell* [15], [16], [18], which adds concurrent threads and synchronizing variables (so-called `MVars`) to Haskell.

All these extensions to the pure part of Haskell give rise to the question whether the extended language has still the nice reasoning properties of the pure functional core language, or put differently: whether the extensions are *safe*. The motivations behind this are manifold: We want to know whether the formal reasoning on purely functional programs we teach in our graduate courses is also sound for real world implementations of Haskell, and whether all the beautiful equations and correctness laws we prove for our tiny and innocent looking functions break in real Haskell as extension of pure core Haskell. Another motivation is to support implementors of Haskell-compilers, aiming at *correctness*. The issue is whether all the program transformations and optimizations implemented for the core part can still be performed for extensions without destroying the semantics of the program.

For the above mentioned extensions of Haskell it is either obvious that they are unsafe (e.g. `unsafePerformIO`) or the situation is not well understood. Moreover, it is also unclear what "safety" of an extension means. For instance, Kiselyov [9] provides an example showing that the extension of pure Haskell by `unsafeInterleaveIO` is not "safe" due to side effects, by refuting conservativity of the extension by lazy file reading. However, there is no consensus on the expressiveness of the counterexample. We exploit the separation between pure functional and impure computations by monadic programming for laying the foundation of correct reasoning, where *conservativity* of an extension is proposed as the "safety" notion, i.e. all the equations that hold in the purely functional core language must also hold after extending the language. A possible alternative approach is to use a precise semantics that models nondeterminism, sharing and laziness (see e.g. [22]) which could be extended to model impure and non-deterministic computations correctly, and then to adapt the compiler accordingly.

As model of Concurrent Haskell we use the (monomorphically) typed process calculus *CHF* which we introduced in [23]. *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
                forkIO (act >>= putMVar ack)
                unsafeInterleaveIO (takeMVar ack)
```

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

Moreover, it is delayed using `unsafeInterleaveIO`. In general, wrapping `unsafeInterleaveIO` around action act_i in `do` $\{x_1 \leftarrow act_1; x_2 \leftarrow act_2; \ldots\}$, breaks the strict sequencing, that is action act_i is performed at the time the value of x_i is *needed* and thus not necessarily before act_{i+1}.

In *CHF* the above `future`-operation is a built-in primitive. Unlike the π-calculus [11], [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 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 $\gg=$ for monadic sequencing. *CHF* is equipped with a monomorphic type system allowing recursive types. In [23] two (semantically equivalent) small-step reduction strategies are introduced 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 one for Concurrent Haskell introduced in [10], [16] where also exceptions are considered. *CHF* also borrows some ideas from the call-by-value lambda calculus with futures [14], [13].

In [23] we showed correctness of several program transformations and that the monad laws hold in *CHF*, under the prerequisite that `seq`'s first argument was restricted to functional types. However, we had to leave open the important question whether the extension of Haskell by concurrency and futures is conservative.

Results. In this paper we address this question and obtain a positive result: *CHF* is a *conservative extension* of its pure sublanguage (Main Theorem 5.5), i.e. the equality 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*. This property is sometimes called *referential transparency*. Haskell's type system is polymorphic with type classes whereas *CHF* has a monomorphic type system. Nevertheless we are convinced that our main result can be transferred to the polymorphic case following our proof scheme, but it would require more (syntactical) effort. Our results also imply that counterexamples like [9] are impossible for *CHF*. We also analyze the boundaries of our conservativity result and show in Section VI that if so-called *lazy futures* [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.

Semantics. As program equivalence for *CHF* we use *contextual equivalence* (following Abramsky [1]): 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 successfully terminate (called *may-convergence*) our notion of contextual equivalence also observes whether a program never looses the ability to may-converge after some reductions (called *should-convergence* or sometimes must-convergence, see e.g. [3], [13], [21], [22]).

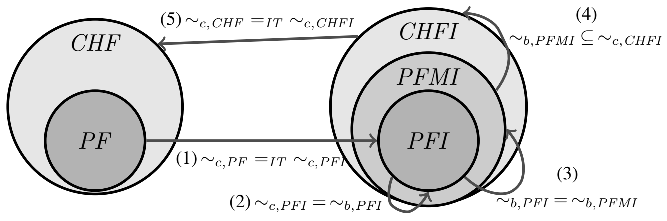
Should vs. Must. Should-convergence slightly differs from the classic notion of must-convergence (e.g. [4]), which additionally requires that all possible computation paths are finite. Some advantages of should-convergence (compared to classical must-convergence) are that restricting the evaluator to *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. Moreover, contextual equivalence has the following invariances: If $e \sim e'$, then e may-converges iff e' may-converges; and e may reach an error iff e' may reach an error, where an error is defined as a program that does not may-converge. Since deadlocks are seen as errors, correct transformations do not introduce errors nor deadlocks in error- and deadlock-free programs, also the non-reachability of an error or deadlock is invariant.

We do not know whether our result also holds for the combination of may- and must-convergence. Technically, a difference shows up in the proof of Proposition 5.3, where the should-convergence proof is possible with only slight modifications of the may-convergence proof, and where it is unclear how to show it for classical must-convergence (or even fair must-convergence).

Consequences. The lessons learned are that there are declarative and also very expressive pure non-strict functional languages with a safe extension by concurrency.

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, e.g. the free theorems in the presence of `seq` [8], or results from call-by-need lambda calculi (e.g. [12], [27]) for reasoning about pure expressions inside Concurrent Haskell.

Proof Technique. Our goal is to show for the pure (deterministic) sublanguage *PF* of *CHF*: any two contextually equivalent *PF*-expressions e_1, e_2 (i.e. $e_1 \sim_{c,PF} e_2$) remain contextually equivalent in *CHF* (i.e. $e_1 \sim_{c,CHF} e_2$). The proof of the main result appears to be impossible by a direct attack. So our proof is indirect and uses the correspondence (see [23]) of the calculus *CHF* with a calculus *CHFI* that unravels recursive bindings into infinite trees and uses call-by-name reduction. The proof structure is illustrated in Fig. 1. Besides *CHFI* there are also sublanguages *PFI* and *PFMI* of *CHFI* which are deterministic and have only expressions, but no processes and `MVars`. While *PFMI* has monadic



$(5)\sim_{c,CHF} =_{IT} \sim_{c,CHFI}$
(4)
$\sim_{b,PFMI} \subseteq \sim_{c,CHFI}$
CHF
$CHFI$
$PFMI$
PF
PFI
$(1)\sim_{c,PF} =_{IT} \sim_{c,PFI}$
(3)
$(2)\sim_{c,PFI} = \sim_{b,PFI}$
$\sim_{b,PFI} = \sim_{b,PFMI}$

Fig. 1. Proof structure

operators, in PFI (like in PF) only pure expressions and types are available. For $e_1 \sim_{c,CHF} e_2$ the corresponding infinite expressions $IT(e_1), IT(e_2)$ (in the calculus PFI) are considered in step (1). Using the results of [23] we are able to show that $IT(e_1)$ and $IT(e_2)$ are contextually equivalent in PFI. In the pure (deterministic) sublanguage PFI of $CHFI$, an applicative bisimilarity $\sim_{b,PFI}$ can be shown to be a congruence, using the method of Howe [6], [7], [20], however extended to infinite expressions. Thus as step (2) we have that $IT(e_1) \sim_{b,PFI} IT(e_2)$ holds. As we show, the bisimilarity transfers also to the calculus $PFMI$ which has monadic operators, and hence we obtain $IT(e_1) \sim_{b,PFMI} IT(e_2)$ (step (3)). This fact then allows to show that both expressions remain contextually equivalent in the calculus $CHFI$ with infinite expressions (step (4)). Finally, in step (5) we transfer the equation $IT(e_1) \sim_{c,CHFI} IT(e_2)$ back to our calculus CHF with finite syntax, where we again use the results of [23].

Outline. In Section II we recall the calculus CHF and introduce its pure fragment PF. In Section III we introduce the calculi $CHFI$, PFI, and $PFMI$ on infinite processes and expressions. We then define applicative bisimilarity for PFI and $PFMI$ in Section IV and show that bisimilarity of PFI and $PFMI$ coincide and also that contextual equivalence is equivalent to bisimilarity in PFI. In Section V we first show that $CHFI$ conservatively extends $PFMI$ and then we go back to the calculi CHF and PF and prove our Main Theorem 5.5 showing that CHF is a conservative extension of PF. In Section VI we show that extending CHF by lazy futures breaks conservativity. Finally, we conclude in Section VII.

For space reasons some proofs are omitted, but can be found in the technical report [24].

II. The CHF-Calculus and its Pure Fragment

We recall the calculus CHF modelling Concurrent Haskell with futures [23]. 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. The syntax of *processes* $Proc_{CHF}$ and expressions $Expr_{CHF}$ is shown in Fig. 2(a). *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 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* which is partitioned into sets, such that each family represents a type constructor T. The data constructors of a type constructor T are ordered, i.e. we write $c_{T,1}, \dots, c_{T,|T|}$, where $|T|$ is the number of constructors belonging to T. We omit the index T, i in $c_{T,i}$ if it is clear from the context. Each $c_{T,i}$ has a fixed arity $\mathrm{ar}(c_{T,i}) \geq 0$. E.g., the type `Bool` has constructors `True` and `False` (both of arity 0) and the type constructor `List` has constructors `Nil` (of arity 0) and `Cons` (of arity 2). We assume that there is a unit type $()$ with constant $()$ as constructor.

Expressions $Expr_{CHF}$ have monadic expressions as a subset (see Fig. 2(a)). Besides the usual constructs of the lambda calculus (variables, abstractions, applications) expressions comprise *constructor applications* $(c\ e_1 \dots e_{\mathrm{ar}(c)})$, `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.

There is a $\texttt{case}_T$-construct for every type constructor T and in `case`-expressions there is a `case`-alternative for every constructor of type constructor T. The variables in a `case`-pattern $(c\ x_1\ \dots\ x_{\mathrm{ar}(c)})$ and also the bound variables in a `letrec`-expression must be pairwise distinct. We sometimes abbreviate the `case`-alternatives as *alts*, i.e. we write $\texttt{case}_T\ e$ `of` *alts*. The expression `return` e is the monadic action which returns e as result, the operator $\gg\!=$ combines two 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 expression 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 by the restriction $\nu x.P$. For the induced notion 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 the *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 the least congruence satisfying the equations shown in Fig. 2(c).

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 shown in Fig. 2(b), where `IO` τ means that an expression of type τ is the result of a monadic action, `MVar` τ is the type of an `MVar`-

$$P, P_i \in \mathit{Proc}_{CHF} ::= P_1 \mid P_2 \mid \nu x.P \mid x \Leftarrow e \mid x = e \mid x\,\mathbf{m}\,e \mid x\,\mathbf{m}\,-$$
$$e, e_i \in \mathit{Expr}_{CHF} ::= x \mid me \mid \lambda x.e \mid (e_1\ e_2) \mid c\ e_1 \ldots e_{\mathrm{ar}(c)}$$
$$\mid \texttt{seq}\ e_1\ e_2 \mid \texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e$$
$$\mid \texttt{case}_T\ e\ \texttt{of}\ alt_{T,1} \ldots\ alt_{T,|T|}$$
$$\text{where } alt_{T,i} = (c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i)$$
$$me \in \mathit{MExpr}_{CHF} ::= \texttt{return}\ e \mid e_1 \ggg= e_2 \mid \texttt{future}\ e$$
$$\mid \texttt{takeMVar}\ e \mid \texttt{newMVar}\ e \mid \texttt{putMVar}\ e_1\ e_2$$

(a) Syntax of Processes, Expressions, and Monadic Expressions

$$\tau, \tau_i \in \mathit{Typ}_{CHF} ::= \texttt{IO}\ \tau \mid (T\ \tau_1 \ldots \tau_n) \mid \texttt{MVar}\ \tau \mid \tau_1 \to \tau_2$$

(b) Syntax of Types

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

(c) Structural Congruence of Processes

$$\mathbb{D} \in \mathit{PCtxt} ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}$$
$$\mathbb{M} \in \mathit{MCtxt} ::= [\cdot] \mid \mathbb{M} \ggg= e$$
$$\mathbb{E} \in \mathit{ECtxt} ::= [\cdot] \mid (\mathbb{E}\ e) \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ alts) \mid (\texttt{seq}\ \mathbb{E}\ e)$$
$$\mathbb{F} \in \mathit{FCtxt} ::= \mathbb{E} \mid (\texttt{takeMVar}\ \mathbb{E}) \mid (\texttt{putMVar}\ \mathbb{E}\ e)$$

(d) Process-, Monadic-, Evaluation-, and Forcing-Contexts

Monadic Computations

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

(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 new thread is not main

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

Closure w.r.t. $\equiv$ and Process Contexts

$$\frac{P \equiv \mathbb{D}[P'], Q \equiv \mathbb{D}[Q'], \text{ and } P' \xrightarrow{CHF} Q'}{P \xrightarrow{CHF} Q}$$

(e) Call-by-name reduction rules of *CHF*

Fig. 2. Syntax, and Semantics of the Calculus *CHF*

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. We omit the (standard) monomorphic typing rules. Special typing restrictions are: (i) $x \Leftarrow e$ is well-typed, if $\Gamma \vdash e :: \texttt{IO}\ \tau$, and $\Gamma \vdash x :: \tau$, (ii) 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 [23]). 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.

A. Operational Semantics and Program Equivalence

In [23] 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. We first introduce some classes of contexts in Fig. 2(d). On the process level there are *process contexts PCtxt*, on expressions first *monadic contexts MCtxt* are used to find the next to-be-evaluated monadic action in a sequence of actions. For the evaluation of (purely functional) expressions usual (call-by-name) *expression evaluation contexts ECtxt* are used, and to enforce the evaluation of the (first) argument of the monadic operators takeMVar and putMVar the class of *forcing contexts FCtxt* is used. A *functional value* is an abstraction or a constructor application, a *value* is a functional value or a monadic expression in *MExpr*.

Definition 2.1: The *call-by-name standard reduction* $\xrightarrow{CHF}$ is defined by the rules and the closure in Fig. 2(e). We 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. For monadic computations 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 finishes 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 in case 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 does 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-may-convergence. Thus we will observe may-convergence and a variant of must-convergence which is called should-convergence (see [21], [22], [23]).

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

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, may-convergent and remains may-convergent under any 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 $\chi \in \{\downarrow_{CHF}, \Downarrow_{CHF}\}$:

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

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 $\forall \mathbb{C}[\cdot^\tau] \in \mathit{Ctxt} : \mathbb{C}[e_1]\chi \implies \mathbb{C}[e_2]\chi$.

B. 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 the expressions Expr_{CHF} where no monadic expression of MExpr_{CHF} is allowed as (sub)-expression. The calculus PF only has *pure types* $\mathit{Typ}_P \subset \mathit{Typ}_{CHF}$, which exclude types which have a subtype of the form $\texttt{IO}\ \tau$ or $\texttt{MVar}\ \tau$. An expression $e \in \mathit{Expr}_{PF}$ is *well-typed with type* $\tau \in \mathit{Typ}_P$ iff $\Gamma \vdash e :: \tau$.

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 $y \overset{\text{main}}{\Longleftarrow} \texttt{seq}\ e\ (\texttt{return}\ ()){\downarrow}_{CHF}$ for some $y \notin FV(e)$. The results in [23] show that convergence does not change if we would have used call-by-need evaluation in CHF (defined in [23]). This allows us 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 contextually equal pure expressions cannot be distinguished in CHF.

III. The Calculi on Infinite Expressions

In this section we introduce three calculi which use *infinite* expressions and we provide the translation IT which translates finite processes Proc_{CHF} into infinite processes and also finite expressions into infinite expressions.

Using the results of [23] we show at the end of this section, that we can perform our proofs in the calculi with infinite expressions before transferring them back to the original calculi CHF and PF with finite syntax.

A. The Calculus CHFI and the Fragments PFMI and PFI

The calculus CHFI (see also [23]) is similar to CHF where instead of finite expressions Expr_{CHF} infinite expressions IExpr_{PFMI} are used, and shared bindings are omitted.

In Fig.3(a) the syntax of infinite monadic expressions IExpr_{PFMI} and infinite processes IProc_{CHFI} is defined, while the former grammar is interpreted co-inductively, the latter is interpreted inductively, but has infinite expressions as subterms. 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, and also S, S_i for infinite processes, and P, P_i for finite processes. Nevertheless, in abuse of notation we will use the same meta symbols for finite as well as infinite contexts.

Compared to finite processes, infinite processes do not comprise shared bindings, but the *silent process* $\mathbf{0}$ is allowed. In infinite expressions `letrec` is not included, but some other special constructs are allowed: The constant `Bot` which represents nontermination and can have any type, and the constants a which are special constants and are available for every type $\texttt{MVar}\ \tau$ for any type $\tau \in \mathit{Typ}_{CHF}$. The calculus CHFI uses the finite types Typ_{CHF} where we assume that in every infinite expression or infinite process every subterm is labeled by its monomorphic type. An infinite expression $s \in \mathit{IExpr}_{PFMI}$ is well-typed with type τ, if $\Gamma \vdash s :: \tau$ cannot be *disproved* by applying the usual monomorphic typing rules. For an infinite process S well-typedness and also well-formedness is defined accordingly. We also use structural congruence $\equiv$ for infinite 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 rules of CHF but adapted to infinite

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

where $alt_{T,i} = (c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to s_i)$ and a are from an infinite set of constants of type $\texttt{MVar}\ \tau$ for every τ

$$
\begin{aligned}
ms \in \mathit{IMExpr}_{PFMI} &::= \texttt{return}\ s \mid s_1\ \texttt{>>=}\ s_2 \mid \texttt{future}\ s\\
&\mid \texttt{takeMVar}\ s \mid \texttt{newMVar}\ s \mid \texttt{putMVar}\ s_1\ s_2\\
S, S_i, \in \mathit{IProc}_{CHFI} &::= S_1 \mid S_2 \mid x \Leftarrow s \mid \nu x.S \mid x\,\mathbf{m}\,s \mid x\,\mathbf{m}\,- \mid \mathbf{0}
\end{aligned}
$$

where $s \in \mathit{IExpr}_{PFMI}$

(a) Syntax of infinite expressions IExpr_{PFI} and infinite processes IProc_{CHFI}

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

(b) Call-by-name reduction rules on infinite expressions of PFI and $PFMI$

The reduction $\xrightarrow{CHFI}$ is assumed to be closed w.r.t. process contexts and structural congruence and $\xrightarrow{CHFI}$ includes the rules (beta), (case), (seq) for functional evaluation and (lunit), (tmvar), (pmvar), (nmvar), (fork) of Fig. 2(e) where the contexts and subexpressions are adapted to infinite expressions and the following reduction rule:

(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 and $/\!/$ means the infinite recursive replacement of s for y.

(c) Standard reduction in $CHFI$

Fig. 3. The Infinite Calculi

expressions performed as infinitary rewriting. We do not list all the reduction rules again, they are analogous to rules for CHF (see Fig. 2(e)), but work on infinite expressions (and adapted contexts) and rule (unIO) is replaced by (unIOTr) which copies the result of a future into all positions. Since in a completely evaluated future $y \Leftarrow \texttt{return}\ s$ the variable y may occur in s this copy operation perhaps must be applied recursively. We formalize this replacement:

Definition 3.1: 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 $x \mapsto \texttt{Bot}$. The operation $/\!/$ is also used for infinite processes with an obvious meaning.

E.g., $(c\,x) \,/\!/\, x$ replaces x by the infinite expression $(c\ (c\ \ldots))$.

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 $S \equiv \nu x_1,\ldots,x_n.(x \overset{\text{main}}{\Longleftarrow} \texttt{return}\ s \mid S')$.

May-convergence $\downarrow_{CHFI}$, *should-convergence* $\Downarrow_{CHFI}$ (and also $\uparrow_{CHFI}$, $\Uparrow_{CHFI}$) as well as contextual equivalence $\sim_{CHFI}$ and contextual preorder $\leq_{CHFI}$ for processes as well as for infinite expressions are defined analogously to CHF where $\xrightarrow{CHFI}$ is used instead of $\xrightarrow{CHF}$.

We also consider the pure fragment of $CHFI$, called the calculus PFI, which has as syntax infinite expressions $\mathit{IExpr}_{PFI} \subset \mathit{IExpr}_{PFMI}$, and contains all infinite expressions of IExpr_{PFMI} that do not have monadic operators ms and also no $\texttt{MVar}$-constants a at any position. As a further calculus we introduce the calculus $PFMI$ which has exactly the set IExpr_{PFMI} as syntax. In $PFMI$ and PFI a *functional value* is an abstraction or a constructor application (except for the constant $\texttt{Bot}$). A *value* of PFI is a functional value and in $PFMI$ a functional value or a monadic expression.

Typing for PFI and $PFMI$ is as explained for $CHFI$ where in the calculus PFI only the pure types Typ_P are available. Standard reduction in PFI and in $PFMI$ is a call-by-name reduction using the rules shown in Fig. 3(b), where $\mathbb{E}$ are call-by-name reduction contexts with infinite expressions as subterms. 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 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.). Contexts ICtxt_{PFMI} (ICtxt_{PFI}, resp.) of $PFMI$ (PFI, resp.) comprise all infinite expressions with a single hole at an expression position.

Definition 3.2: Contextual equivalence in PFI is defined as $\sim_{c,PFI} := \leq_{c,PFI} \cap \geq_{c,PFI}$ where for $s,t :: \tau$ the inequation $s \leq_{c,PFI} t$ holds if, and only if $\forall \mathbb{C}[\cdot :: \tau] \in \mathit{ICtxt}_{PFI} : \mathbb{C}[s]\downarrow_{PFI} \implies \mathbb{C}[t]\downarrow_{PFI}$.

As a further notation we introduce the set IExpr^c_{PFMI} (IExpr^c_{PFI}, resp.) as the set of *closed* infinite expressions of IExpr_{PFMI} (IExpr_{PFI}, resp.).

B. The Translation IT

We will now use a translation from [23] which translates CHF-processes into $CHFI$-processes by removing `letrec`- and shared bindings. It is known that the translation does not change the convergence behavior of processes.

Definition 3.3 ([23]): For a process P the translation $IT :: \mathit{Proc} \to \mathit{IProc}$ translates 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 $\mathbf{0}$-component. Free variables, futures, and names of MVars are kept in the tree (are not replaced). Equivalence of infinite processes is syntactic, where we do not distinguish α-equal trees. Similarly, IT is also defined for expressions to translate PFI-expressions into PF-expressions.

Theorem 3.4 ([23]): For all processes $P \in \mathit{Proc}_{CHF}$: ($P\downarrow_{CHF}$ iff $IT(P)\downarrow_{CHFI}$) and ($P\Downarrow_{CHF}$ iff $IT(P)\Downarrow_{CHFI}$).

An analogous result can also be derived for the pure fragments of *CHF* and *CHFI*:

Proposition 3.5: For all *PF*-expressions e_1, e_2 it holds: $e_1 \leq_{c,PF} e_2$ iff $IT(e_1) \leq_{c,PFI} IT(e_2)$.

Proof: An easy consequence of Theorem 3.4 is 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 $IT(\cdot)$-images of *PF*-expressions. 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 $\mathbb{C}[IT(e_2)]{\downarrow_{PFI}}$. ■

As the next step we will show that *CHFI* conservatively extends *PFI*. Theorem 3.4 and Proposition 3.5 will then enable us to conclude that *CHF* conservatively extends *PF*.

IV. Simulation in the Calculi *PFI* and *PFMI*

We will now consider a simulation relation in the two calculi *PFI* and *PFMI*. Using Howe's method it is possible to show that both similarities are precongruences. We will then show that *PFMI* extends *PFI* conservatively w.r.t. similarity.

A. Similarities in PFMI and PFI are Precongruences

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 4.1: 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 4.2: 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 4.3: 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 $s, t \in \mathit{IExpr}^c_{PFMI}$ 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 4.4 (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 4.3 and Proposition 4.4 can be transferred to *PFI*, where we use $\leq_{b,PFI}$ and F_{PFI} as notation. Determinism of $\xrightarrow{PFMI}$ implies:

Lemma 4.5: If $s \xrightarrow{PFMI} s'$, then $s' \leq^o_{b,PFMI} s \wedge s \leq^o_{b,PFMI} s'$.

In [24, Theorem B.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 4.6: $\leq^o_{b,PFMI}$ is a precongruence on infinite expressions IExpr_{PFMI} and $\leq^o_{b,PFI}$ is a precongruence on infinite expressions IExpr_{PFI}. If σ is a substitution, then $s \leq^o_{b,PFMI} t$ implies $\sigma(s) \leq^o_{b,PFMI} \sigma(t)$ and also $s \leq^o_{b,PFI} t$ implies $\sigma(s) \leq^o_{b,PFI} \sigma(t)$.

B. Behavioral and Contextual Preorder in PFI

We now investigate the relationships between the behavioral and contextual preorders in the two calculi *PFI* and *PFMI* of infinite expressions. We 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.

Lemma 4.7: $\leq^o_{b,PFI}\ \subseteq\ \leq_{c,PFI}$.

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 4.8: 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 $(\leq_{c,PFI})^c$ satisfies the fixpoint condition, i.e. $(\leq_{c,PFI})^c \subseteq F_{PFI}((\leq_{c,PFI})^c)$: Let s, t be closed and $s \leq_{c,PFI} t$. If $s{\downarrow_{PFI}}\theta(s_1, \dots, s_n)$, then also $t{\downarrow_{PFI}}$. Using the appropriate `case`-expressions as contexts, it is easy to see that $t{\downarrow_{PFI}}\theta(t_1, \dots, t_n)$. Now we have to show that $s_i \leq^o_{c,PFI} t_i$. 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 4.7 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 4.7 and 4.8 imply:

Theorem 4.9: 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 V for the language *PFMI* the following technical lemma on $\leq^o_{b,PFI}$ is required.

Lemma 4.10: 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 $s_2[s_1 \,/\!/\, x] \leq^o_{b,PFMI} t_2[t_1 \,/\!/\, x]$.

C. Behavioral Preorder in PFMI

We now 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 4.11: 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. The types are translated by replacing all $(\texttt{IO}\ \tau)$ and $(\texttt{MVar}\ \tau)$-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 4.12: For all closed *PFMI*-expressions s it holds:
(1) $s{\downarrow_{PFMI}}$ if, and only if $\phi(s){\downarrow_{PFI}}$.
(2) If $s{\downarrow_{PFMI}}\theta(s_1, \dots, s_n)$ then $\phi(s){\downarrow_{PFI}}\phi(\theta(s_1, \dots, s_n))$.
(3) 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 4.13: $\leq_{b,PFI}\ \subseteq\ \leq_{b,PFMI}$.

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 4.14: 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 4.13. ■

Proposition 4.15: Let x be a variable of type $(\texttt{MVar}\ \tau)$ 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. ■

V. Conservativity of *PF* in *CHF*

In this section we will first show that $s \leq^o_{b,PFMI} t$ implies $s \leq_{c,CHFI} t$ and then we transfer the results back to the calculi with finite expressions and processes and derive our main theorem.

A. Conservativity of PFMI in CHFI

We will 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 $\mathbb{C}[\cdot_\tau]$ with an expression hole and $s, t :: \tau$.

In the following, we drop the distinction between MVar-constants and variables. 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 $\mathbb{G} \in \mathit{GCtxt}$ 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. We will first consider *GCtxt*-contexts and later lift the result to all contexts of *CHFI*.

Proposition 5.1: Let $s_i, t_i :: \tau_i$ be *PFMI*-expressions with $s_i \leq^o_{b,PFMI} t_i$, and let $\mathbb{G}[\cdot_{\tau_1}, \dots, \cdot_{\tau_n}] \in \mathit{GCtxt}$. Then $\mathbb{G}[s_1, \dots, s_n]{\downarrow_{CHFI}} \implies \mathbb{G}[t_1, \dots, t_n]{\downarrow_{CHFI}}$.

Proof: Let $\mathbb{G}[s_1, \dots, s_n]{\downarrow_{CHFI}}$. We use induction on the number of reductions of $\mathbb{G}[s_1, \dots, s_n]$ to a successful process. In the base case $\mathbb{G}[s_1, \dots, s_n]$ is successful. Then either $\mathbb{G}[t_1, \dots, t_n]$ is also successful, or $\mathbb{G} = \nu\mathcal{X}.x \overset{\text{main}}{\Longleftarrow} [\cdot] \mid \mathbb{G}'$, 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 $\mathbb{G}[t_1, \dots, t_n]$ to a successful process.

Now let $\mathbb{G}[s_1, \dots, s_n] \xrightarrow{CHFI} S_1$ 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 $s_1 = \mathbb{M}_1[\mathtt{return}\ r_1 \gg\!= r_2]$, 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 $\mathbb{M}_{1,j} = [\cdot] \gg\!= s'_j$. By induction on the depth, there is a reduction sequence $t_1 \xrightarrow{CHFI,*} \mathbb{M}_{2,1}\ldots\mathbb{M}_{2,k}[t'_1 \gg\!= t'_2]$, where $\mathbb{M}_{2,j} = [\cdot] \gg\!= r'_j$, $s'_j \leq^o_{b,PFMI} r'_j$, and $\mathtt{return}\ r_1 \leq^o_{b,PFMI} t'_1$. Let $\mathbb{M}_2 := \mathbb{M}_{2,1}\ldots\mathbb{M}_{2,k}$. This implies $t'_1 \xrightarrow{CHFI} \mathtt{return}\ t''_1$ 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 4.15 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 be increased.

For (unIOTr) as argued above, $\mathbb{G}[t_1,\ldots,t_n]$ can be reduced such that also a (unIOTr) reduction is applicable. Assume that the substitutions are $\sigma_s = s' /\!/ x$ and $\sigma_t = t' /\!/ x$ for $\mathbb{G}[s_1,\ldots,s_n]$ and the reduction-successor of $\mathbb{G}[t_1,\ldots,t_n]$. Lemma 4.10 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 5.2: Let $s := \mathtt{Bot}$, $t := \mathtt{takeMVar}\ x$, and $\mathbb{G}[\cdot] := z \overset{\mathsf{main}}{\Longleftarrow} \mathtt{takeMVar}\ x \mid y \Leftarrow [\cdot] \mid x\,\mathbf{m}\,e$. 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 5.3: 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 and show that $\mathbb{G}[t_1,\ldots,t_n]\uparrow_{CHFI}$ 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 5.1 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 $t_1 = \mathbb{M}[\mathtt{return}\ r_1 \gg\!= r_2]$, 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 $s_1 \xrightarrow{*} \mathbb{M}_1[\mathtt{return}\ r'_1 \gg\!= r'_2]$. 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 5.1. For (unIOTr), Lemma 4.10 shows that for the substitutions $\sigma := s /\!/ x$ and $\sigma' := s' /\!/ x$ 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. ■

Theorem 5.4: Let $s, t \in$ *IExpr*$_{PFMI}$ with $s \sim^o_{b,PFMI} t$. Then $s \sim_{c,CHFI} t$.

Proof: Let $s, t \in$ *IExpr*$_{PFMI}$ with $s \sim^o_{b,PFMI} t$. We only show $s \leq_{c,CHFI} t$ since the other direction follows by symmetry. We first consider may-convergence: Let $\mathbb{C}$ be a process context of *CHFI* with an expression hole such that $\mathbb{C}[s]\downarrow_{CHFI}$. Let $\mathbb{C} = \mathbb{C}_1[\mathbb{C}_2]$ such that $\mathbb{C}_2$ is the maximal expression context. Then $\mathbb{C}_2[s] \sim_{b,PFMI} \mathbb{C}_2[t]$ since $\sim_{b,PFMI}$ is a congruence. Since $\mathbb{C}_1$ is a *GCtxt*-context, Proposition 5.1 implies $\mathbb{C}_1[\mathbb{C}_2[t]]\downarrow_{CHFI}$, i.e. $\mathbb{C}[t]\downarrow_{CHFI}$. Showing $\mathbb{C}[s]\Downarrow_{CHFI} \implies \mathbb{C}[t]\Downarrow_{CHFI}$ follows by the same reasoning using Proposition 5.3. ■

B. The Main Theorem: Conservativity of PF in CHF

We now 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.

Main Theorem 5.5: 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 3.5 shows that $e_1 \sim_{c,PF} e_2$ is equivalent to $IT(e_1) \sim_{c,PFI} IT(e_2)$. Now Theorem 4.9 and Proposition 4.14 show that $IT(e_1) \sim_{b,PFMI} IT(e_2)$. Then Theorem 5.4 shows that $IT(e_1) \sim_{c,CHFI} IT(e_2)$. Finally, from Theorem 3.4 it easily follows that $e_1 \sim_{c,CHF} e_2$. ■

VI. 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 $\mathtt{seq}\ e_1\ e_2$ and $\mathtt{seq}\ e_2\ (\mathtt{seq}\ e_1\ e_2)$ used by Kiselyov's counterexample [9], holds in the pure calculus and in *CHF* (see [24, Proposition C.2]). This implies that Kiselyov's counterexample 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:

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

The rule (lfork) creates a lazy future. Evaluation can turn a lazy future into a concurrent future if its value is demanded by 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 following context $\mathbb{C}$ that uses lazy futures and distinguishes the two expressions:

$$\begin{aligned}\mathbb{C} = {} & x \overset{lazy}{\Longleftarrow} \texttt{takeMVar}\ v \gg\!= \lambda w.\mathbb{C}_1[v] \\ & \mid y \overset{lazy}{\Longleftarrow} \texttt{takeMVar}\ v \gg\!= \lambda w.\mathbb{C}_1[v] \mid v\,\mathbf{m}\,\texttt{True} \\ & \mid z \overset{\text{main}}{\Longleftarrow} \texttt{case}\ [\cdot]\ \texttt{of}\ (\texttt{True} \rightarrow \bot)\ (\texttt{False} \rightarrow \texttt{return True}) \\ \mathbb{C}_1 = {} & (\texttt{putMVar}\ [\cdot]\ \texttt{False} \gg\!= \lambda\_ \rightarrow \texttt{return}\ w)\end{aligned}$$

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

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)
```

VII. 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 [23]. Future work is to rigorously show that our results can be extended to polymorphic typing, (simple monomorphising does not work due to recursive lets). We also will analyze further extensions like killing threads, and synchronous and asynchronous exceptions (as in [10], [16]), where our working hypothesis is that killing threads and (at least) synchronous exceptions retain our conservativity result.

References

[1] S. Abramsky, "The lazy lambda calculus," in *Research Topics in Functional Programming*. Addison-Wesley, 1990, pp. 65–116.

[2] H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," in *Symposium on Artificial intelligence and programming languages*. ACM, 1977, pp. 55–59.

[3] A. Carayol, D. Hirschkoff, and D. Sangiorgi, "On the representation of McCarthy's amb in the pi-calculus." *Theoret. Comput. Sci.*, vol. 330, no. 3, pp. 439–473, 2005.

[4] R. De Nicola and M. Hennessy, "Testing equivalences for processes," *Theoret. Comput. Sci.*, vol. 34, pp. 83–133, 1984.

[5] R. H. Halstead, Jr., "Multilisp: a language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 501–538, 1985.

[6] D. Howe, "Equality in lazy computation systems," in *LICS'89*, IEEE Press, 1989, pp. 198–203.

[7] ——, "Proving congruence of bisimulation in functional programming languages," *Inform. and Comput.*, vol. 124, no. 2, pp. 103–112, 1996.

[8] P. Johann and J. Voigtländer, "The impact of seq on free theorems-based program transformations," *Fund. Inform.*, vol. 69, no. 1–2, pp. 63–102, 2006.

[9] O. Kiselyov, "Lazy IO breaks purity," 2009, Haskell Mailinglist, http://haskell.org/pipermail/haskell/2009-March/021064.html.

[10] S. Marlow, S. L. P. Jones, A. Moran, and J. H. Reppy, "Asynchronous exceptions in haskell," in *PLDI*, 2001, pp. 274–285.

[11] R. Milner, *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[12] K. Nakata and M. Hasegawa, "Small-step and big-step semantics for call-by-need," *J. Funct. Program.*, vol. 19, pp. 699–722, 2009.

[13] 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.*, vol. 173, pp. 313–337, 2007.

[14] J. Niehren, J. Schwinghammer, and G. Smolka, "A concurrent lambda calculus with futures," *Theoret. Comput. Sci.*, vol. 364, no. 3, pp. 338–356, 2006.

[15] S. Peyton Jones, A. Gordon, and S. Finne, "Concurrent Haskell," in *POPL'96*, ACM, 1996, pp. 295–308.

[16] S. Peyton Jones, "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell," in *Engineering theories of software construction*. IOS-Press, 2001, pp. 47–96.

[17] ——, *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[18] S. Peyton Jones and S. Singh, "A tutorial on parallel and concurrent programming in haskell," in *AFP'08*. Springer, 2009, pp. 267–305.

[19] S. Peyton Jones and P. Wadler, "Imperative functional programming," in *POPL'93*. ACM, 1993, pp. 71–84.

[20] A. M. Pitts, "Howe's method for higher-order languages," in *Advanced Topics in Bisimulation and Coinduction*, Cambridge University Press, 2011, vol. 52, pp. 197–232.

[21] A. Rensink and W. Vogler, "Fair testing," *Inform. and Comput.*, vol. 205, no. 2, pp. 125–198, 2007.

[22] 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.*, vol. 18, no. 03, pp. 501–553, 2008.

[23] ——, "A contextual semantics for Concurrent Haskell with futures," in *PPDP '11*. ACM, 2011, pp. 101–112.

[24] ——, "On conservativity of Concurrent Haskell," Inst. f. Informatik, Goethe-Universität Frankfurt am Main, Frank report 47, 2012, http://www.ki.cs.uni-frankfurt.de/papers/frank/.

[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.*, vol. 110, no. 6, pp. 232 – 235, 2010.

[27] M. Schmidt-Schauß, M. Schütz, and D. Sabel, "Safety of Nöcker's strictness analysis," *J. Funct. Programming*, vol. 18, no. 04, pp. 503–551, 2008.

[28] P. Sestoft, "Deriving a lazy abstract machine," *J. Funct. Programming*, vol. 7, no. 3, pp. 231–264, 1997.

[29] H. Søndergaard and P. Sestoft, "Referential transparency, definiteness and unfoldability," *"Acta Inform.*, vol. 27, pp. 505–517, 1989.

[30] P. Wadler, "Monads for functional programming," in *AFP'95*. LNCS 925. Springer, 1995, pp. 24–52.