Correctness of an STM Haskell Implementation

Manfred Schmidt-Schauss and David Sabel

Goethe-University, Frankfurt, Germany

Technical Report Frank-50

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 23, 2012

Abstract. A concurrent implementation of software transactional memory in Concurrent Haskell using a call-by-need functional language with processes and futures is given. The description of the small-step operational semantics is precise and explicit, and employs an early abort of conflicting transactions. A proof of correctness of the implementation is given for a contextual semantics with may- and should-convergence. This implies that our implementation is a correct evaluator for an abstract specification equipped with a big-step semantics.

1 Introduction

Due to the recent development in hardware and software, concurrent and parallel programming is getting more and more important, and thus there is a need for semantical investigations in concurrent programming. In this paper we investigate programming models of concurrent processes (threads) where several processes may access shared memory. Concurrent access and independency of processes lead to the well-known problems of conflicting memory use and thus requires protection of critical sections to ensure mutual exclusion. Over the years several programming primitives like locks, semaphores, monitors, etc. have been introduced and used to assure this atomicity of memory operations in a concurrent setting. However, the explicit use of locking mechanisms is error-prone – the programmer may omit to set or release a lock resulting in deadlocks or race conditions – and it is also often inefficient, since setting too many locks may sequentialize program execution and prohibit concurrent evaluation. Another obstacle of lock-based concurrency is that composing larger programs from smaller ones is usually impossible [5].

A recent approach to overcome these difficulties is software transactional memory (STM) [13,14,5,6] where operations on the shared memory are viewed as *transactions* on the memory, i.e. several small operations (like read and write of memory locations) are compound to a transaction and then the system (the transaction manager) ensures that the transaction is performed in an atomic, isolated, and consistent manner. I.e. the programmer can assume that all the transactions are performed sequentially and isolated, while the runtime system may perform them concurrently and interleaved (invisible to the programmer). This removes the burden from the programmer to set locks and to keep track of them. Composability of transactions is another very helpful feature of STM. We focus on the Haskell-based approach in [5,6], which provides even more advantages like separation of different kinds of side effects (`IO` and `STM`) by monadic programming and the type system of Haskell. Memory-transactions are indicated in the program by marking a sequence of reductions as atomic.

Though STM is easy to use for the programmer, implementing a *correct* transaction manager is considerably harder. Hurdles are an exact specification of the correctness properties, of course to provide an implementation, and to prove its correctness and the validity of the properties. In this paper we will address and solve these problems. We start with a language and calculus *SHF* (STM-Haskell with futures) that is rather close to STM-Haskell [5,6] and shares some ideas with the concurrent call-by-need process calculus investigated in [10,11]. In difference to STM-Haskell threads are modelled by futures, *SHF* has no exceptions, it uses call-by-need evaluation and bindings in the environment for sharing, and – for simplicity – it is equipped with a monomorphic type system. We keep the monadic modelling for the separation of `IO` and `STM` and the composability, in particular the selection composability by `orElse`. A big-step operational semantics is given for *SHF*, which is obviously correct, since transactions are executed in isolation, and their effect on the shared memory are only observable after a successful execution. However, this semantics is *not implementable*, since for instance it requires to solve the halting problem. Thus the purpose of defining *SHF* is not the implementation but the *specification* of a correct STM-system.

Secondly, we define a concurrent implementation of *SHF* by introducing the concurrent calculus *CSHF*. *CSHF* is close to a real implementation, since its operational semantics is formulated as a detailed, precise and complete small-step reduction semantics where all precautions and retries of the transactions are explicitly represented, and with appropriate granularity of the small-step reductions. Features of *CSHF* are registration of threads at TVars and forced aborts of transactions in case of conflicts. All applicability conditions for the reductions are decidable. *CSHF* is designed to enable concurrent (i.e. interleaved) execution of threads as much as possible, i.e. there are only very short phases where internal locking mechanisms are used to prevent race conditions.

The main goal of our investigation is to show that the concurrent implementation fulfills the specification. Here we use the strong criterion of contextual equivalence w.r.t. may- as well as should-convergence in both calculi, where may-convergence means that a process has the ability to evaluate to a successful process, and should-convergence means that the ability to become successful is never lost on every reduction path. Observing also should-convergence for contextual equivalence ensures that it is not permitted to transform a program P that cannot reach an error-state (i.e. a state that is not may-convergent) into a program that can reach an error-state. In Main Theorem 4.17 we obtain the important result that the implementation mapping ψ is *observationally correct* [12], i.e. for every context D and process P in *SHF* it holds: $D[P]$ may-converges (or should-converges, resp.), if, and only if $\psi(D)[\psi(P)]$ is may-convergent (should-convergent) in *CSHF*. Observational correctness thus shows that may- and should-convergence is preserved and reflected by the implementation ψ (i.e. ψ is convergence equivalent) and the tests used for contextual equivalence are translated in a compositional way. A direct consequence is also that ψ is *adequate*, i.e. the implementation preserves all contextual inequalities and thus does not introduce new equalities (which would introduce confusion in the implementation). Note that even the proof of convergence equivalence of ψ is a strong result, since it implies that *CSHF* is a correct evaluator for *SHF*.

Our notion of correctness is very strong, since it implies the properties of the concurrent program that are aimed at in other papers like atomicity, opacity, asf. A surprising consequence is that early abortion of conflicting transaction is *necessary* to make the implementation correct, where “early” means that a committing transaction must abort other conflicting transactions. Comparing our results with the implementation in [6], our work is a justification for the correctness of most of the design decisions of their implementation.

2 The SHF-Calculus

The syntax of *SHF* and its *processes Proc* is in Fig. 1(a).
We assume a countable infinite set of variables (denoted by $x, y, z, \ldots$) and a countable infinite set of *identifiers* (denoted by u, u') to identify threads. In a *parallel composition* $P_1 \mid P_2$ the processes P_1 and P_2 run concurrently, and the *name restriction* $\nu x.P$ restricts the scope of x to process P. A *concurrent thread* $\langle u \wr x \rangle \Leftarrow e$ has *identifier* u and evaluates the expression e binding the result to the variable x (called the *future* x). A process has usually one distinguished thread, the *main thread*, denoted by $\langle u \wr x \rangle \overset{\mathsf{main}}{\Longleftarrow} e$. Evaluation of the main thread is enforced, which does not hold for other threads. *TVars* $x\,\mathbf{t}\,e$ are the transactional (mutable) variables with name x and content e. They can only be accessed inside STM-transactions. *Bindings* $x = e$ model globally shared expressions, where we call x a *binding variable*. Futures and names of TVars are called *component-names*. A variable x is an *introduced variable* if it is a binding variable or a component-name. A process is *well-formed*, if all introduced variables and identifiers are pairwise distinct, and it has at most one main thread.

We assume a partitioned set of *data constructors*, where each family represents a type constructor T. The T-data constructors are ordered (denoted with $c_1, \ldots, c_{|T|}$). Each type constructor T and each data constructor c has an arity $\mathrm{ar}(T) \geq 0$ ($\mathrm{ar}(c) \geq 0$, resp.). The functional language has variables, *abstractions* $\lambda x.e$, and *applications* $(e_1\ e_2)$, *constructor applications* $(c\ e_1 \ldots e_{\mathrm{ar}(c)})$, `case`-*expressions* where for every type constructor T there is one $\mathtt{case}_T$-construct.

$P, P_i \in \mathit{Proc} ::= P_1 \mid P_2 \mid \nu x.P \mid \langle u \wr x \rangle \Leftarrow e \mid x = e \mid x\,\mathbf{t}\,e$

$e, e_i \in \mathit{Expr} ::= x \mid m \mid \lambda x.e \mid (e_1\ e_2) \mid c\ e_1 \ldots e_{\mathrm{ar}(c)} \mid \texttt{seq}\ e_1\ e_2$
$\quad \mid \texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e$
$\quad \mid \texttt{case}_T\ e\ \texttt{of}\ \mathit{alt}_{T,1} \ldots\ \mathit{alt}_{T,|T|}$ where $\mathit{alt}_{T,i} = (c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i)$

$m \in \mathit{MExpr} ::= \texttt{return}_{\texttt{IO}}\ e \mid e_1 \gg\!\!=_{\texttt{IO}} e_2 \mid \texttt{future}\ e \mid \texttt{atomically}\ e \mid \texttt{return}_{\texttt{STM}}\ e$
$\quad \mid e_1 \gg\!\!=_{\texttt{STM}} e_2 \mid \texttt{retry} \mid \texttt{orElse}\ e_1\ e_2 \mid \texttt{newTVar}\ e \mid \texttt{readTVar}\ e \mid \texttt{writeTVar}\ e$

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

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

Functional values: abstractions $\lambda x.e$ and constructor applications $c\ e_1\ \ldots\ e_n$
Monadic values: all monadic expressions $m \in \mathit{MExpr}$
Values: functional values and monadic values
cx-values: $(c\ x_1 \ldots x_n)$ and monadic values where all arguments are variables

(b) Functional values, monadic values, cx-values, and values

$\mathbb{D} \in \mathit{PC} ::= [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D}$
$\mathbb{E} \in \mathit{EC} ::= [\cdot] \mid (\mathbb{E}\ e) \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ \mathit{alts}) \mid (\texttt{seq}\ \mathbb{E}\ e)$
$\mathbb{M}_{\texttt{IO}} \in \mathit{MC}_{\texttt{IO}} ::= [\cdot] \mid \mathbb{M}_{\texttt{IO}} \gg\!\!=_{\texttt{IO}} e$
$\mathbb{M}_{\texttt{STM}} \in \mathit{MC}_{\texttt{STM}} ::= \mathbb{M}_{\texttt{IO}}[\texttt{atomically}\ \widehat{\mathbb{M}}_{\texttt{STM}}]$
$\quad$ where $\widehat{\mathbb{M}}_{\texttt{STM}} \in \widehat{\mathit{MC}_{\texttt{STM}}} ::= [\cdot] \mid \widehat{\mathbb{M}}_{\texttt{STM}} \gg\!\!=_{\texttt{STM}} e \mid \texttt{orElse}\ \widehat{\mathbb{M}}_{\texttt{STM}}\ e$
$\mathbb{F} \in \mathit{FC} ::= \mathbb{E} \mid (\texttt{readTVar}\ \mathbb{E}) \mid (\texttt{writeTVar}\ \mathbb{E}\ e)$
$\mathbb{L}_Q \in \mathit{LC}_Q ::= \langle u \wr x \rangle \Leftarrow \mathbb{M}_Q[\mathbb{F}]$ where $Q \in \{\texttt{IO}, \texttt{STM}\}$
$\quad \mid \langle u \wr x \rangle \Leftarrow \mathbb{M}_Q[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1$ where $\mathbb{E}_1 \neq [\cdot]$

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

$(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3) \quad P_1 \mid P_2 \equiv P_2 \mid P_1 \quad \nu x_1.\nu x_2.P \equiv \nu x_2.\nu x_1.P$
$(\nu x.P_1) \mid P_2 \equiv \nu x.(P_1 \mid P_2)$ if $x \notin FV(P_2) \qquad P_1 \equiv P_2$ if $P_1 =_\alpha P_2$

(d) Structural Congruence

Fig. 1. The calculus *SHF*, syntax and contexts

We may abbreviate `case`-*alternatives* with *Alts*. For $\texttt{case}_T$ there is exactly one alternative $(c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i)$ for every constructor $c_{T,i}$ of type constructor T. In a *pattern* $c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})}$ the variables x_i must be pairwise distinct. In the alternative $(c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i)$ the variables x_i become bound with scope e_i. Further constructs are `seq`-*expressions* $(\texttt{seq}\ e_1\ e_2)$ for strict evaluation, and `letrec`-*expressions* to implement local sharing and recursive bindings. In $\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e$ the variables x_i must be pairwise distinct and the bindings $x_i = e_i$ are recursive, i.e. the scope of x_i is $e_1, \ldots, e_n$ and e.

Monadic expressions comprise variants for the IO- and the STM-monad of the "bind" operator `>>=` for sequential composition of actions, and the `return`-operator. For the STM-monad `newTVar`, `readTVar`, and `writeTVar` are available to create and access TVars, the primitive `retry` to abort and restart the STM-transaction, and $\texttt{orElse}\ e_1\ e_2$ to compose an `STM`-transaction from e_1, e_2: `orElse` returns if e_1 is successful and if it catches a `retry` in e_1 then it proceeds with e_2. For the IO-monad the `future`-operator creates threads, and `atomically` lifts an STM-transaction into the IO-monad by executing the transaction.

Variable binders are introduced by abstractions, `letrec`, `case`-alternatives, and $\nu x.P$. This induces a notion of free and bound variables and α-renaming and α-equivalence (denoted by $=_\alpha$). Let $FV(P)$ ($FV(e)$, resp) be the free variables of process P (expression e, resp.). We assume the *distinct variable convention* to hold, and also assume α-renaming is implicitly performed, if necessary.

A *context* is a process or an expression with a hole (denoted by $[\cdot]$). We write $C[e]$ ($C[P]$, resp.) for filling the hole of context C by expression e (process P, resp.). For processes we use a *structural congruence* $\equiv$ to equate obviously equal processes, which is the least congruence satisfying the equations of Fig. 1(d).

SHF is equipped with a monomorphic type system. However, we "overload" the constructors and the monadic operators by assuming that they have polymorphic type according to the usual conventions, however in the language they are used as monomorphic. The polymorphic types of the monadic operators are:

$\mathtt{future} :: (\mathtt{IO}\ \alpha) \to \mathtt{IO}\ \alpha$ $\qquad \mathtt{return}_{\mathtt{IO}} :: \alpha \to \mathtt{IO}\ \alpha$
$\mathtt{atomically} :: (\mathtt{STM}\ \alpha) \to \mathtt{IO}\ \alpha$ $\qquad (\gg=_{\mathtt{IO}}) :: (\mathtt{IO}\ \alpha_1) \to (\alpha_1 \to \mathtt{IO}\ \alpha_2) \to \mathtt{IO}\ \alpha_2$
$\mathtt{return}_{\mathtt{STM}} :: \alpha \to \mathtt{STM}\ \alpha$ $\qquad (\gg=_{\mathtt{STM}}) :: (\mathtt{STM}\ \alpha_1) \to (\alpha_1 \to \mathtt{STM}\ \alpha_2) \to \mathtt{STM}\ \alpha_2$
$\mathtt{readTVar} :: (\mathtt{TVar}\ \alpha) \to \mathtt{STM}\ \alpha$ $\qquad \mathtt{writeTVar} :: (\mathtt{TVar}\ \alpha) \to \alpha \to \mathtt{STM}\ ()$
$\mathtt{newTVar} :: \alpha \to \mathtt{STM}(\mathtt{TVar}\ \alpha)$ $\qquad \mathtt{orElse} :: (\mathtt{STM}\ \alpha) \to (\mathtt{STM}\ \alpha) \to (\mathtt{STM}\ \alpha)$
$\mathtt{retry} :: (\mathtt{STM}\ \alpha)$

The syntax of monomorphic types *Typ* is given in Fig. 1(a), where $(\mathtt{IO}\ \tau)$ means a monadic IO-action with return type τ, $(\mathtt{STM}\ \tau)$ means an STM-transaction action, $(\mathtt{TVar}\ \tau)$ stands for a `TVar`-reference with content type τ, $(T\ \tau_1\ \ldots\ \tau_n)$ is a type for an n-ary type constructor T, and $\tau_1 \to \tau_2$ is a function type. To fix the types during reduction and for transformations, we assume that every variable x is explicitly typed and thus has a built-in type $\Gamma(x)$. For contexts, we assume that the hole $[\cdot]$ is typed and carries a type label. The notation $\Gamma \vdash e :: \tau$ ($\Gamma \vdash P :: \mathtt{wt}$, resp.) means that expression e (process P, resp.) can be typed with type τ (can be typed, resp.) using Γ. We omit the full set of typing rules (see Appendix A), but note that $\langle u \wr x \rangle \Leftarrow e$ is well-typed if x is of type τ and e is of type $\mathtt{IO}\ \tau$, and that an `STM`- or an `IO`-type for the first argument of `seq` is forbidden, to enable that the monad laws hold (see [11]).

Definition 2.1. *A process P is* well-typed *iff P is well-formed and $\Gamma \vdash P :: \mathtt{wt}$ holds. An expression e is* well-typed with type τ *iff $\Gamma \vdash e :: \tau$ holds.*

We define a *small-step reduction relation* $\xrightarrow{SHF}$ (called *standard reduction*) for *SHF* with two intermediate big-step style reductions for the atomic execution of an STM-transaction (i.e. the evaluation of e in `atomically` e and of `orElse` $e\ e'$). The intermediate reduction of transactions is called $\xrightarrow{SHFA}$.

Definition 2.2. *A well-formed process P is* successful, *if P has a main thread of the form $\langle u \wr x \rangle \overset{\mathsf{main}}{\Longleftarrow} \mathtt{return}\ e$, i.e. $P \equiv \nu x_1.\ldots.\nu x_n.(\langle u \wr x \rangle \overset{\mathsf{main}}{\Longleftarrow} \mathtt{return}\ e \mid P')$.*

Definition 2.3. *The* standard reduction rules *are given in Fig. 2 where the used contexts are defined in Fig. 1(c). Standard reductions are permitted only for well-formed and non-successful processes.*

Monadic STM Computations:

(lunit$_{\mathtt{STM}}$) $\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[\mathtt{return}_{\mathtt{STM}}\ e_1 \mathbin{\mathtt{>\!\!>=}_{\mathtt{STM}}}\ e_2] \xrightarrow{SHFA} \langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[e_2\ e_1]$

(read) $\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[\mathtt{readTVar}\ x] \mid x\,\mathbf{t}\,e$
$\xrightarrow{SHFA} \nu z.(\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[\mathtt{return}_{\mathtt{STM}}\ z] \mid z = e \mid x\,\mathbf{t}\,z)$

(write) $\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[\mathtt{writeTVar}\ x\ e_2] \mid x\,\mathbf{t}\,e_1 \xrightarrow{SHFA} \langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[\mathtt{return}_{\mathtt{STM}}\ ()] \mid x\,\mathbf{t}\,e_2$

(nvar) $\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[\mathtt{newTVar}\ e] \xrightarrow{SHFA} \nu x.(\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[\mathtt{return}_{\mathtt{STM}}\ x] \mid x\,\mathbf{t}\,e)$

(ortry)
$$\frac{\nu X.\mathbb{D}[\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[\mathtt{orElse}\ e_1\ e_2]] \xrightarrow{SHFA,*} \nu X.\mathbb{D}'[\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[\mathtt{orElse}\ \mathtt{retry}\ e_2]]}{\nu X.\mathbb{D}[\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[\mathtt{orElse}\ e_1\ e_2]] \xrightarrow{SHFA} \nu X.\mathbb{D}[\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[e_2]]}$$

(orret) $\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[\mathtt{orElse}\ (\mathtt{return}_{\mathtt{STM}} e_1)\ e_2] \xrightarrow{SHFA} \langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[e_1]$

(retryup) $\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[\mathtt{retry} \mathbin{\mathtt{>\!\!>=}_{\mathtt{STM}}}\ e_1] \xrightarrow{SHFA} \langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{STM}}[\mathtt{retry}]$

Monadic IO Computations:

(lunit$_{\mathtt{IO}}$) $\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{IO}}[\mathtt{return}_{\mathtt{IO}}\ e_1 \mathbin{\mathtt{>\!\!>=}_{\mathtt{IO}}}\ e_2] \xrightarrow{SHF} \langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{IO}}[e_2\ e_1]$

(fork) $\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{IO}}[\mathtt{future}\ e] \xrightarrow{SHF} \nu z, u'.(\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{IO}}[\mathtt{return}\ z] \mid \langle u' \wr z\rangle \Leftarrow e)$
where z, u' are fresh and the created thread is not the main thread

(unIO) $\langle u \wr y\rangle \Leftarrow \mathtt{return}\ e \xrightarrow{SHF} y = e$ if the thread is not the main-thread

(atomic)
$$\frac{\nu X.\mathbb{D}_1[\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically}\ e]] \xrightarrow{SHFA,*} \nu X.\mathbb{D}_1'[\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically}\ (\mathtt{return}_{\mathtt{STM}}\ e')]]}{\mathbb{D}[\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically}\ e]] \xrightarrow{SHF} \mathbb{D}'[\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{IO}}[\mathtt{return}_{\mathtt{IO}}\ e']]}$$
where $\mathbb{D} = \nu X.(\mathbb{D}_1 \mid \mathbb{D}_2)$ and $\mathbb{D}_1$ contains all bindings and TVars of $\mathbb{D}$, $\mathbb{D}_2$ contains all futures of $\mathbb{D}$ and where $\mathbb{D}' = \nu X.(\mathbb{D}_1' \mid \mathbb{D}_2)$

Functional Evaluation:

(feval$_{\mathtt{IO}}$) $P \xrightarrow{SHF} P'$, if $P \xrightarrow{a} P'$ for $a \in \{\mathrm{cp}_{\mathtt{IO}}, \mathrm{abs}_{\mathtt{IO}}, \mathrm{mkb}_{\mathtt{IO}}, \mathrm{lbeta}_{\mathtt{IO}}, \mathrm{case}_{\mathtt{IO}}, \mathrm{seq}_{\mathtt{IO}}\}$

(feval$_{\mathtt{STM}}$) $P \xrightarrow{SHFA} P'$, if $P \xrightarrow{a} P$ for $a \in \{\mathrm{cp}_{\mathtt{STM}}, \mathrm{abs}_{\mathtt{STM}}, \mathrm{mkb}_{\mathtt{STM}}, \mathrm{lbeta}_{\mathtt{STM}}, \mathrm{case}_{\mathtt{STM}}, \mathrm{seq}_{\mathtt{STM}}\}$

The reductions with parameter $Q \in \{\mathtt{STM}, \mathtt{IO}\}$ are defined as follows:

(cp$_Q$) $\mathbb{L}_Q[x_1] \mid x_1 = x_2 \mid \ldots \mid x_{n-1} = x_n \mid x_n = v$
$\rightarrow \mathbb{L}_Q[v] \mid x_1 = x_2 \mid \ldots \mid x_{n-1} = x_n \mid x_n = v$,
if v is an abstraction, a cx-value or a component-name

(abs$_Q$) $\mathbb{L}_Q[x_1] \mid x_1 = x_2 \mid \ldots \mid x_{m-1} = x_m \mid x_m = c\ e_1 \ldots\ e_n$
$\rightarrow \nu y_1, \ldots y_n.(\mathbb{L}_Q[c\ y_1\ \ldots\ y_n] \mid x_1 = x_2 \mid \ldots \mid x_{m-1} = x_m$
$\mid x_m = c\ y_1\ \ldots\ y_n \mid y_1 = e_1 \mid \ldots \mid y_n = e_n)$
if c is a constructor, or $\mathtt{return}_{\mathtt{STM}}$, $\mathtt{return}_{\mathtt{IO}}$, $\mathtt{>\!\!>=}_{\mathtt{STM}}$, $\mathtt{>\!\!>=}_{\mathtt{IO}}$, `orElse`, `atomically`, `readTVar`, `writeTVar`, `newTVar`, or `future` and $n \geq 1$, and some e_i is not a variable.

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

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

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

(case$_Q$) $\mathbb{L}_Q[\mathtt{case}_T\ c\ \mathtt{of}\ \ldots(c \rightarrow e)\ldots] \rightarrow \mathbb{L}_Q[e]$

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

Closure:
$$\frac{P_i \equiv D[P_i'], P_1' \xrightarrow{SHF} P_2'}{P_1 \xrightarrow{SHF} P_2} \qquad \frac{P_i \equiv D[P_i'], P_1' \xrightarrow{SHFA} P_2'}{P_1 \xrightarrow{SHFA} P_2}$$

Fig. 2. The calculus *SHF*, reductions

In the following we will denote the transitive closure of a relation $\xrightarrow{R}$ by $\xrightarrow{R,+}$ and the reflexive-transitive closure by $\xrightarrow{R,*}$.

We define the $\xrightarrow{SHF}$-*redex*: For (lunit), (fork), it is the expression in the context $\mathbb{M}$, for (unIO), it is $\langle u \wr y \rangle \Leftarrow \texttt{return}\ e$, for (mkb), (lbeta), (case), (seq), (cp), (abs) it is the expression (or variable) in the context $\mathbb{L}$. We define the $\xrightarrow{SHFA}$-*redex*: For (lunit$_{\texttt{STM}}$), (read), (write), (nvar), (ortry), (orret) it is the expression in the context $\mathbb{M}_{\texttt{STM}}$, for (mkb$_{\texttt{STM}}$), (lbeta$_{\texttt{STM}}$), (case$_{\texttt{STM}}$), (seq$_{\texttt{STM}}$), (cp$_{\texttt{STM}}$), (abs$_{\texttt{STM}}$) it is the expression (or variable) in the context $\mathbb{L}_{\texttt{STM}}$.

We explain the standard reduction rules of Fig. 2. The rule (lunit) implements the semantics of the monadic sequencing operator $\gg=$. The rules (read), (write), and (nvar) access and create TVars. The rule (ortry) is a big-step rule: if a $\xrightarrow{SHFA,*}$-reduction sequence starting with `orElse` e_1 e_2 ends in `orElse retry` e_2, then the effects are ignored, and `orElse` e_1 e_2 is replaced by e_2. If the reduction of e_1 ends with `return` e, then rule (orret) is used to keep the result as the result of `orElse` e_1 e_2. The rule (atomic) is also a big-step rule. If for a single thread the $\xrightarrow{SHFA,*}$-reduction successfully produces a return, then the transaction is performed in one step of the $\xrightarrow{SHF}$-reduction. If the $\xrightarrow{SHFA,*}$-reduction ends in a `retry`, in a stuck expressions or does not terminate, then there is no $\xrightarrow{SHF}$-reduction, and hence it is omitted from the operational semantics. The rule (fork) spawns a new concurrent thread and returns the newly created future as the result. The rule (unIO) binds the result of a monadic computation to a functional binding, i.e. the value of a concurrent future becomes accessible.

The rules (cp) and (abs) inline a needed binding $x = e$ where e must be an abstraction, a cx-value, or a component name. To implement call-by-need evaluation the arguments of constructor applications and monadic expressions are shared by new bindings, similar to lazy copying [15]. Since the variable (binding-) chains are transparent, there is no need to copy binding-variables to other places in the expressions. The rule (mkb) moves `letrec`-bindings into the global bindings. The rule (lbeta) is the sharing variant of β-reduction. The (case)-reduction reduces a `case`-expression, where perhaps bindings are created to implement sharing. The (seq)-rule evaluates a `seq`-expression.

Since the reduction rules only introduce variables which are fresh and never introduce a main thread, $\xrightarrow{SHF}$ preserves well-formedness. Also type preservation holds, since every redex keeps the type of subexpressions.

Contextual equivalence equates processes P_1, P_2 if their observable behavior is indistinguishable if P_1 and P_2 are plugged into any process context. For non-deterministic (and concurrent) calculi observing may-convergence, i.e. whether a process can be reduced to a successful process, is *not* sufficient and thus we will observe may-convergence and should-convergence (see [8,9]).

Definition 2.4. *A process P* may-converges *(written as $P{\downarrow}$), iff it is well-formed and reduces to a successful process, (see Definition 2.2), i.e. $P{\downarrow}$ iff P is well-formed and $\exists P' : P \xrightarrow{SHF,*} P' \wedge P'$ successful. If $P{\downarrow}$ does not hold, then P* must-diverges *written as $P{\Uparrow}$. A process P* should-converges *(writ-*

ten as $P{\Downarrow}$), iff it is well-formed and remains may-convergent under reduction, i.e. $P{\Downarrow}$ iff P is well-formed and $\forall P' : P \xrightarrow{SHF,*} P' \implies P'{\downarrow}$. If P is not should-convergent then we say P may-diverges *written as $P{\uparrow}$, which is also equivalent to $\exists P' : P \xrightarrow{SHF,*} P' \wedge P'{\Uparrow}$.*

Contextual approximation $\leq_c$ *and* contextual equivalence $\sim_c$ *on processes are defined as $\leq_c := \leq_{\downarrow} \cap \leq_{\Downarrow}$ and $\sim_c := \leq_c \cap \geq_c$ where for $\zeta \in \{\downarrow, \Downarrow\}$: $P_1 \leq_\zeta P_2$ iff $\forall \mathbb{D} \in PC : \mathbb{D}[P_1]\zeta \implies \mathbb{D}[P_2]\zeta$.*

The definition of $\xrightarrow{SHF}$ implies that non-wellformed processes are always must-divergent. Also, the process construction by $\mathbb{D}[P]$ is always well-typed if P is well-typed, since we assume that variables have a built-in type.

3 A Concurrent Implementation of STM Evaluation

While *SHF* obviously implements STM in a correct manner, since transactions are performed isolated and atomically, there are two drawbacks for implementing this semantics: There is few concurrency, and the rules (atomic) and (ortry) have undecidable preconditions, since they include the halting problem. That is why we introduce a concurrent (small-step) evaluation enabling much more concurrency which is closer to an abstract machine than the big-step semantics. The executability of every single step is decidable, and every state has a finite set of potential successors. The undecidable conditions in the rules (atomic) and (ortry) are checked by tentatively executing the transaction, under concurrency, thus allowing other threads to execute. Transaction execution should guarantee an equivalent linearized execution of transactions. Instead of locking methods, we look for lock-free transaction handling. Instead of using an optimistic read/write approach which performs a rollback in case of a conflict [4], we will follow a pessimistic read and write: there are no locks at the start of a transaction, reads and writes are local, only at the end of a successful transaction there is a commit phase and updates become visible to other transactions where a real, but internal, locking is used for a short time. To have a correct overall execution, conflicting transactions will be stopped by sending them a retry-notification (so-called early conflict detection), where the knowledge of the potential conflicts is memorized at the TVars.

We describe our variant of concurrent execution of software transactional memory including lock-free transactions, where our ultimate goal is to show that our concurrent variant is *correct*, i.e. to show that it is semantically equivalent to the big-step reduction defined for *SHF*. The main idea is to view a transaction as a function $V_1 \to V_2$ – from a set of input TVars V_1 to a set of modified TVars V_2 where V_1, V_2 may have common elements. The guarantee must be that at the end of transaction execution, the complete transaction could be atomically and instantaneously executed on the TVars V_1, V_2. At the end of the transaction, the set of read TVars and the set of updated TVars must be memorized at the executing thread, and other transactions that have read any variable of V_2, but are not finished yet, need to be aborted (restarted) due to a conflict. Sharing by bindings is carefully maximized, including transparent variable-variable-binding chains, which leads to manageable correctness proofs.

Now we detail on this idea and introduce the calculus *CSHF* which has a concurrent evaluation of transactions and slightly adapts the syntax of *SHF*. First we describe the syntax changes of the language, and then exactly describe the rules, which are close to a concurrent abstract machine for SHF.

Definition 3.1. *The syntax of CSHF-processes is almost the same as for SHF-processes where, however, there are some extensions and changes.*

Instead of TVars $x\,\mathbf{t}\,e$ *there are two constructs: The* global *TVars are* $x\,\mathbf{tg}\,e\,u\,g$, *where the additional third argument* u *is a locking label and may be empty (written as* $\emptyset$*) or a thread identifier* u *that locks the TVar, and* g *is a list of thread identifiers for those threads that want to be notified for a retry, when* x *is updated. A* stack of thread-local *TVars:* $u\,\mathbf{tls}\,s$, *where* u *is a thread identifier and* s *is a stack of sets with elements* $x\,\mathbf{tl}\,e$, *where* x *is a name of a TVar, and* e *is an expression.*

A thread may have a transaction log*, which is only available if a thread is within a transaction. It is written over the thread-arrow as* $\langle u \wr y\rangle \xLeftarrow{T,L;K} e$ *where* T *is a set of TVars (i.e. the names of the TVars) that are read during the transaction;* L *is a stack of triples* (L_a, L_n, L_w) *where* L_a *is a set of the names of* $\underline{a}$*ll TVars which are accessed during the transaction,* L_n *a set of names of* $\underline{n}$*ewly generated TVars,* L_w *a set of locally updated (or* $\underline{w}$*ritten) TVars, and the stack reflects the depth of the* `orElse`*-execution;* K *is a set of TVar-names that is locked by a thread in the commit-phase.*

Additional variants of the operators `orElse` *and* `atomically` *are required:* `orElse!` *indicates that* `orElse` *is active, and* `atomically!` *that a transaction is active.* `atomically!` *has as a second argument an expression that is a saved copy of the start expression and that will again be activated after rollback and restart. I.e., the sets of monadic expressions and* $MC_{\mathtt{STM}}$*-contexts are adapted as:*

$$\begin{aligned} m \in MExpr ::= {} & \mathtt{return}_{\mathtt{IO}}\, e \mid e_1 \gg\!=_{\mathtt{IO}} e_2 \mid \mathtt{future}\, e \mid \mathtt{return}_{\mathtt{STM}}\, e \mid e_1 \gg\!=_{\mathtt{STM}} e_2 \\ & \mid \mathtt{atomically}\, e \mid \mathtt{atomically!}\, e\, e' \mid \mathtt{retry} \mid \mathtt{orElse}\, e_1\, e_2 \mid \mathtt{orElse!}\, e_1\, e_2 \\ & \mid \mathtt{newTVar}\, e \mid \mathtt{readTVar}\, e \mid \mathtt{writeTVar}\, e \end{aligned}$$

$$\begin{aligned} \mathbb{M}_{\mathtt{STM}} \in MC_{\mathtt{STM}} ::= {} & \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ \widehat{\mathbb{M}}_{\mathtt{STM}}\ e] \\ & \textit{where } \widehat{\mathbb{M}}_{\mathtt{STM}} \in \widehat{MC_{\mathtt{STM}}} ::= [\cdot] \mid \widehat{\mathbb{M}}_{\mathtt{STM}}\ \gg\!=_{\mathtt{STM}}\ e \mid \mathtt{orElse!}\ \widehat{\mathbb{M}}_{\mathtt{STM}}\ e \end{aligned}$$

A thread that corresponds to the syntax of SHF-processes is called non-transactional*, and a thread that uses one of the syntax components* `orElse!`*,* `atomically!`*, or a transaction log is called* transactional*. A CSHF-process that only has non-transactional threads and no* $u\,\mathbf{tls}\,s$*-components is called* non-transactional*; otherwise, it is called* transactional*.*

We say a thread is currently *performing a transaction*, if the current evaluation focusses on the arguments of `atomically!` (see the reduction $\xrightarrow{CSHF}$ below).

Definition 3.2. *A CSHF-process P is* well-formed *iff the following holds: Variable names of TVars, threads, and binders are introduced at most once;*

i.e. threads, bindings, and global TVars are unique per variable. For every component $x\,\mathbf{tl}\,e$ in a stack-entry of $u\,\mathbf{tls}\,s$, either there is also a global TVar $x\,\mathbf{tg}\,e\,o\,g$, or the TVar is in the L_n-component of the thread-memory (the locally generated TVars). Moreover, for every thread identifier u, there is at most one process component $u\,\mathbf{tls}\,s$. In every stack entry, names of TVars occur at most once. For every thread identifier u in $u\,\mathbf{tls}\,s$ there exists a thread with this identifier.

Though the syntax of *CSHF* is slightly different from *SHF*, we use the same names for the context classes $PC, EC, MC_{\mathtt{STM}}, MC, FC, MC_{\mathtt{IO}}$, and LC_Q. If necessary, then we distinguish the context classes using an index C (for concurrent). For the PC-contexts we assume that also transactional threads are permitted.

Definition 3.3 (Operational Semantics of ***CSHF*****).** *A well-formed CSHF-process P reduces to another CSHF-process P' (denoted by $P \xrightarrow{CSHF} P'$) iff $P \equiv D[P_1]$ and $P' \equiv D[P_1']$ and $P_1 \to P_1'$ by a reduction rule in Fig. 3, 4, and 5.*

We explain the execution of a transaction and the use of the transaction log, where we also point to the rules of the operational semantics. When the execution is started a new empty transaction log is created (atomic).
Read-Operation: A read first looks into the local store. If no local TVar exists, then the global value is copied into the local store and the own thread identifier is added to the notify-list of the global TVar ((readl) and (readg)).
Write-Operation: A write command always writes into the local store, perhaps preceded by a copy from the global into the local store ((writel) and (writeg)).
OrElse-Evaluation: If evaluation descends into the left expression of `orElse`, then the stacks of TVar-names and of local TVars are extended by duplicating their top element (orElse). For the final write in the commit phase only the top of the stack is relevant. If evaluation of the left expression is successful, the stack remains as it is (orReturn). In case of a retry, the top element is popped (orRetry) and the execution of the second expression then uses the stack before executing the `orElse`. Note that the information on the read TVars is kept in the set T, since the values may have an influence on the outcome of the `orElse`-tree execution. This is necessary, since the big-step semantics of *SHF* enforces a left-to-right evaluation of the `orElse`-tree, where the leftmost successful try will be kept. Note that this is semantically different from making a non-deterministic choice of one of the successful possibilities in the `orElse`-tree.
Commit-Phase: At the end of a transaction, there is a lock-protected sequence of updates: A thread that is in its commit-phase, first locks all its globally read and to-be-updated TVars (writeStart). Locked variables cannot be accessed by other threads for reading, or modifying the notify-list. Then all own notification-entries are removed (writeClear). All the threads, which are unequal to the running thread, and that are in the notify-list of the updated TVars, will be stopped by replacing the current transaction expression by `retry` (sendRetry). This mechanism is like raising (synchronous) exceptions to influence other threads. The to-be-updated TVars are written into the global store (writeTV), then the locks are released (unlockTV) and fresh TVars are also moved to the global store (writeTVn). Finally, the transaction log is removed (writeEnd).

Monadic IO Computations:

(lunit$_{\texttt{IO}}$) $\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\texttt{IO}}[\texttt{return}_{\texttt{IO}}\ e_1 \gg\!=_{\texttt{IO}}\ e_2] \xrightarrow{CSHF} \langle u \wr y\rangle \Leftarrow \mathbb{M}_{\texttt{IO}}[e_2\ e_1]$

(fork) $\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\texttt{IO}}[\texttt{future}\ e] \xrightarrow{CSHF} \nu z, u'.(\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\texttt{IO}}[\texttt{return}\ z] \mid \langle u' \wr z\rangle \Leftarrow e)$
where z, u' are fresh and the created thread is not the main thread

(unIO) $\langle u \wr y\rangle \Leftarrow \texttt{return}\ e \xrightarrow{SHF} y = e$ if the thread is not the main-thread

Monadic STM Computations:

(atomic) $\langle u \wr y\rangle \Leftarrow \mathbb{M}_{\texttt{IO}}[\texttt{atomically}\ e]$
$\xrightarrow{CSHF} \nu z.\langle u \wr y\rangle \xLeftarrow{\emptyset,[\emptyset]} \mathbb{M}_{\texttt{IO}}[\texttt{atomically!}\ z\ z] \mid u\,\textbf{tls}\,[\emptyset] \mid z = e$

(lunit$_{\texttt{STM}}$) $\langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[\texttt{return}_{\texttt{STM}}\ e_1 \gg\!=_{\texttt{STM}}\ e_2] \xrightarrow{CSHF} \langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[e_2\ e_1]$

(readl) $\langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[\texttt{readTVar}\ x] \mid u\,\textbf{tls}\,(\{x\,\textbf{tl}\,e_1\} \dot\cup r) : s$
$\xrightarrow{CSHF} \nu z.(\langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[\texttt{return}_{\texttt{STM}}\ z] \mid z = e_1 \mid u\,\textbf{tls}\,(\{x\,\textbf{tl}\,z\} \dot\cup r) : s$

(readg) $\langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[\texttt{readTVar}\ x] \mid x\,\textbf{tg}\,e_1\,\emptyset\,g \mid u\,\textbf{tls}\,r : s \xrightarrow{CSHF}$
$\nu z.(\langle u \wr y\rangle \xLeftarrow{T',L'} \mathbb{M}_{\texttt{STM}}[\texttt{return}_{\texttt{STM}}\ z] \mid z = e_1 \mid u\,\textbf{tls}\,(\{x\,\textbf{tl}\,z\} \dot\cup r) : s \mid x\,\textbf{tg}\,z\,\emptyset\,g')$
if $x \notin L_a$ where $L = (L_a, L_n, L_w) : L_r$, $L' = (L_a \cup \{x\}, L_n, L_w) : L_r$,
$T' = T \cup \{x\}$ and $g' = (\{u\} \cup g)$

(writel) $\langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[\texttt{writeTVar}\ x\ e_1] \mid u\,\textbf{tls}\,(\{x\,\textbf{tl}\,e_2\} \dot\cup r : s)$
$\xrightarrow{CSHF} \langle u \wr y\rangle \xLeftarrow{T,L'} \mathbb{M}_{\texttt{STM}}[\texttt{return}_{\texttt{STM}}\ ()] \mid u\,\textbf{tls}\,(\{x\,\textbf{tl}\,e_1\} \dot\cup r : s)$
where $L = (L_a, L_n, L_w) : L_r$, $L' = (L_a, L_n, L_w \cup \{x\}) : L_r$

(writeg) $\langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[\texttt{writeTVar}\ x\ e_1] \mid x\,\textbf{tg}\,e_2\,\emptyset\,g \mid u\,\textbf{tls}\,(r : s)$
$\xrightarrow{CSHF} \langle u \wr y\rangle \xLeftarrow{T,L'} \mathbb{M}_{\texttt{STM}}[\texttt{return}_{\texttt{STM}}\ ()] \mid x\,\textbf{tg}\,e_2\,\emptyset\,g \mid u\,\textbf{tls}\,(\{x\,\textbf{tl}\,e_1\} \dot\cup r : s)$
if $x \notin L_a$, where $L = (L_a, L_n, L_w) : L_r$, $L' = (L_a \cup \{x\}, L_n, L_w \cup \{x\}) : L_r$

(nvar) $\langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[\texttt{newTVar}\ e] \mid u\,\textbf{tls}\,(r : s)$
$\xrightarrow{CSHF} \nu x.(\langle u \wr y\rangle \xLeftarrow{T,L'} \mathbb{M}_{\texttt{STM}}[\texttt{return}_{\texttt{STM}}\ x] \mid u\,\textbf{tls}\,(\{x\,\textbf{tl}\,e)\} \dot\cup r : s)$
where $L = (L_a, L_n, L_w)$ and $L' = (\{x\} \cup L_a, \{x\} \cup L_n, L_w)$

(retryup) $\langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[\texttt{retry} \gg\!=_{\texttt{STM}}\ e_1] \xrightarrow{CSHF} \langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[\texttt{retry}]$

(orElse) $\langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[\texttt{orElse}\ e_1\ e_2] \mid (u\,\textbf{tls}\,(\{x_1\,\textbf{tl}\,e_{1,1}, \dots, x_n\,\textbf{tl}\,e_{1,n}\} : s)$
$\xrightarrow{CSHF} \nu z_1, \dots z_n.(\langle u \wr y\rangle \xLeftarrow{T,L'} \mathbb{M}_{\texttt{STM}}[\texttt{orElse!}\ e_1\ e_2]$
$\mid u\,\textbf{tls}\,((\{x_1\,\textbf{tl}\,z_1, \dots, x_n\,\textbf{tl}\,z_n\} : (\{x_1\,\textbf{tl}\,z_1, \dots, x_n\,\textbf{tl}\,z_n\}) : s)$
$\mid z_1 = e_{1,1} \mid \dots \mid z_n = e_{1,n})$
where $L = (L_a, L_n, L_w) : L_r$, $L' = (L_a, L_n, L_w) : ((L_a, L_n, L_w) : L_r)$

(orRetry) $\langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[\texttt{orElse!}\ \texttt{retry}\ e_2] \mid u\,\textbf{tls}\,(r : s)$
$\xrightarrow{CSHF} \langle u \wr y\rangle \xLeftarrow{T,L'} \mathbb{M}_{\texttt{STM}}[e_2] \mid u\,\textbf{tls}\,s$ where $L = L_e : L'$

(orReturn) $\langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[\texttt{orElse!}\ (\texttt{return}\ e_1)\ e_2] \mid u\,\textbf{tls}\,(r : s)$
$\xrightarrow{CSHF} \langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{STM}}[e_1] \mid u\,\textbf{tls}\,(r : s)$

(retryCGlob) $\langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\texttt{IO}}[\texttt{atomically!}\ \texttt{retry}\ e] \mid x\,\textbf{tg}\,e_1\,\emptyset\,g$
$\xrightarrow{CSHF} \langle u \wr y\rangle \xLeftarrow{T',L} \mathbb{M}_{\texttt{IO}}[\texttt{atomically!}\ \texttt{retry}\ e] \mid x\,\textbf{tg}\,e_1\,\emptyset\,g'$
if $x \in T \neq \emptyset$, where $T' = T \setminus \{x\}$, $g' = g \setminus \{u\}$

(retryEnd) $\langle u \wr y\rangle \xLeftarrow{\emptyset,L} \mathbb{M}_{\texttt{IO}}[\texttt{atomically!}\ \texttt{retry}\ e] \mid u\,\textbf{tls}\,(r : s)$
$\xrightarrow{CSHF} \langle u \wr y\rangle \Leftarrow \mathbb{M}_{\texttt{IO}}[\texttt{atomically}\ e]$

Fig. 3. Concurrent Implementation of *SHF*, transaction reductions

The commit-phase uses thread-local memory K, written over the thread-arrow after a semicolon. The third memory-component is the set of locked TVars.

(writeStart) start of commit: locking the read and to-be-written TVars

$\langle u \wr y\rangle \xLeftarrow{T,L} \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ (\mathtt{return}_{\mathtt{STM}}\ e_1)\ e] \mid x_1\,\mathbf{tg}\,e'_1\,\emptyset\,g_1 \mid \ldots \mid x_n\,\mathbf{tg}\,e'_n\,\emptyset\,g_n$

$\xrightarrow{CSHF} \langle u \wr y\rangle \xLeftarrow{T,L;K} \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ (\mathtt{return}_{\mathtt{STM}}\ e_1)\ e] \mid x_1\,\mathbf{tg}\,e'_1\,u\,g_1 \mid \ldots \mid x_n\,\mathbf{tg}\,e'_n\,u\,g_n$

where $K := \{x_1,\ldots,x_n\} = T \cup (L_a \setminus L_n)$ and $L = (L_a, L_n, L_w) : L_r$

(writeClear) removing the notify-entries of u:

$\langle u \wr y\rangle \xLeftarrow{T,L;K} \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ (\mathtt{return}_{\mathtt{STM}}\ e_1)\ e] \mid x\,\mathbf{tg}\,e_2\,u\,g$

$\xrightarrow{CSHF} \langle u \wr y\rangle \xLeftarrow{T',L;K} \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ (\mathtt{return}_{\mathtt{STM}}\ e_1)\ e] \mid x\,\mathbf{tg}\,e_2\,u\,g'$

if $x \in T$ where $g' = g \setminus \{u\}$ and $T' = T \setminus \{x\}$

(sendRetry) Sending other threads (that are in transactions) a retry:

$\langle u \wr y\rangle \xLeftarrow{\emptyset,L;K} \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ (\mathtt{return}_{\mathtt{STM}}\ e_1)\ e] \mid x\,\mathbf{tg}\,e_2\,u\,g$

$\quad \mid \langle u' \wr z\rangle \xLeftarrow{T',L'} \mathbb{M}'_{\mathtt{IO}}[\mathtt{atomically!}\ e_3\ e_4]$

$\xrightarrow{CSHF} \langle u \wr y\rangle \xLeftarrow{\emptyset,L;K} \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ (\mathtt{return}_{\mathtt{STM}}\ e_1)\ e] \mid x\,\mathbf{tg}\,e_2\,u\,g'$

$\quad \mid \langle u' \wr z\rangle \xLeftarrow{T',L'} \mathbb{M}'_{\mathtt{IO}}[\mathtt{atomically!}\ \mathtt{retry}\ e_4]$

if $x \in L_w$, $g \neq \emptyset$, $u' \in g$, where $L = (L_a, L_n, L_w) : L_r$, and $g = g' \mathbin{\dot\cup} \{u'\}$

(writeTV) Overwriting the global TVars with the local TVars of u:

$\langle u \wr y\rangle \xLeftarrow{\emptyset,L;K} \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ (\mathtt{return}_{\mathtt{STM}}\ e_1)\ e] \mid x\,\mathbf{tg}\,e_2\,u\,\emptyset \mid u\,\mathbf{tls}\,(\{x\,\mathbf{tl}\,e_3\} \mathbin{\dot\cup} r : s)$

$\xrightarrow{CSHF} \langle u \wr y\rangle \xLeftarrow{\emptyset,L';K} \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ (\mathtt{return}_{\mathtt{STM}}\ e_1)\ e] \mid x\,\mathbf{tg}\,e_3\,u\,\emptyset \mid u\,\mathbf{tls}\,(r : s)$

if for all $z \in L_w \setminus L_n$ the g is empty in the component $z\,\mathbf{tg}\,e_2\,u\,g$, and if $L_w \setminus L_n \neq \emptyset$ where $L = (L_a, L_n, L_w) : L_r$, $x \in L_w \setminus L_n$, $L' = (L_a, L_n, L_w \setminus \{x\}) : L_r$.

(unlockTV) Unlocking the locked TVars:

$\langle u \wr y\rangle \xLeftarrow{\emptyset,L;K} \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ (\mathtt{return}_{\mathtt{STM}}\ e_1)\ e] \mid x\,\mathbf{tg}\,e_2\,u\,\emptyset$

$\xrightarrow{CSHF} \langle u \wr y\rangle \xLeftarrow{\emptyset,L;K'} \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ (\mathtt{return}_{\mathtt{STM}}\ e_1)\ e] \mid x\,\mathbf{tg}\,e_3\,\emptyset\,\emptyset$

if $L_w \setminus L_n = \emptyset$, and $K \neq \emptyset$ where $L = (L_a, L_n, L_w) : L_r$, and $K' = K \setminus \{x\}$

(writeTVn) Moving the freshly generated TVars of u into global store:

$\langle u \wr y\rangle \xLeftarrow{\emptyset,L;\emptyset} \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ (\mathtt{return}_{\mathtt{STM}}\ e_1)\ e] \mid u\,\mathbf{tls}\,(\{x\,\mathbf{tl}\,e_3\} \mathbin{\dot\cup} r : s)$

$\xrightarrow{CSHF} \langle u \wr y\rangle \xLeftarrow{\emptyset,L;\emptyset} \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ (\mathtt{return}_{\mathtt{STM}}\ e_1)\ e] \mid x\,\mathbf{tg}\,e_3\,\emptyset\,\emptyset$

if $L_n \neq \emptyset$, $x \in L_n$, where $L = (L_a, L_n, L_w) : L_r$ and $L' = (L_a, L_n \setminus \{x\}, \emptyset) : L_r$

(writeEnd) Removing local store and end of commit of transaction:

$\langle u \wr y\rangle \xLeftarrow{\emptyset,L;\emptyset} \mathbb{M}_{\mathtt{IO}}[\mathtt{atomically!}\ (\mathtt{return}_{\mathtt{STM}}\ e_1)\ e] \mid u\,\mathbf{tls}\,(r : s)$

$\xrightarrow{CSHF} \langle u \wr y\rangle \Leftarrow \mathbb{M}_{\mathtt{IO}}[\mathtt{return}_{\mathtt{IO}}\ e_1]$

if no other rules of the commit-phase for u are applicable, i.e., if $L_n = L_w = \emptyset$, where $L = (L_a, L_n, L_w) : L_r$.

Fig. 4. Concurrent Implementation of *SHF*, commit phase of transaction

Functional Evaluation

(feval$_{\texttt{IO}}$) $P \xrightarrow{CSHF} P'$, if $P \xrightarrow{a} P'$ for $a \in \{\mathrm{cp}_{\texttt{IO}}, \mathrm{abs}_{\texttt{IO}}, \mathrm{mkb}_{\texttt{IO}}, \mathrm{lbeta}_{\texttt{IO}}, \mathrm{case}_{\texttt{IO}}, \mathrm{seq}_{\texttt{IO}}\}$

(feval$_{\texttt{STM}}$) $P \xrightarrow{CSHF} P'$, if $P \xrightarrow{a} P'$ for $a \in \{\mathrm{cp}_{\texttt{STM}}, \mathrm{abs}_{\texttt{STM}}, \mathrm{mkb}_{\texttt{STM}}, \mathrm{lbeta}_{\texttt{STM}}, \mathrm{case}_{\texttt{STM}}, \mathrm{seq}_{\texttt{STM}}\}$

The reductions with parameter $Q \in \{\texttt{STM}, \texttt{IO}\}$ are defined as follows

(cp$_Q$) $\mathbb{L}_Q[x_1] \mid x_1 = x_2 \mid \ldots \mid x_{n-1} = x_n \mid x_n = v$
$\rightarrow \mathbb{L}_Q[v] \mid x_1 = x_2 \mid \ldots \mid x_{n-1} = x_n \mid x_n = v$,
if v is an abstraction, a cx-value, or a component name

(abs$_Q$) $\mathbb{L}_Q[x_1] \mid x_1 = x_2 \mid \ldots \mid x_{m-1} = x_m \mid x_m = c\ e_1 \ldots\ e_n$
$\rightarrow \nu y_1, \ldots y_n.(\mathbb{L}_Q[c\ y_1\ \ldots\ y_n] \mid x_1 = x_2 \mid \ldots \mid x_{m-1} = x_m$
$\mid x_m = c\ y_1\ \ldots\ y_n \mid y_1 = e_1 \mid \ldots \mid y_n = e_n)$
if c is a constructor, or $\texttt{return}_{\texttt{STM}}$, $\texttt{return}_{\texttt{IO}}$, $\texttt{>>=}_{\texttt{STM}}$, $\texttt{>>=}_{\texttt{IO}}$, `orElse`, `atomically`, `readTVar`, `writeTVar`, `newTVar`, or `future`, and $n \geq 1$, and some e_i is not a variable.

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

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

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

(case$_Q$) $\mathbb{L}_Q[\texttt{case}_T\ c\ \texttt{of}\ \ldots(c \rightarrow e)\ldots] \rightarrow \mathbb{L}_Q[e]$

(seq$_Q$) $\mathbb{L}_Q[(\texttt{seq}\ v\ e)] \rightarrow \mathbb{L}_Q[e]$, if v is a functional value

Fig. 5. Concurrent implementation of *SHF*, functional reductions

Rollback and Restart: A transaction is rolled back and restarted by the `retry`-command (if it is not inside an `orElse`-command). This can occur by a user programmed `retry`, or if the transaction gets stopped by a conflicting transaction which is committing. The thread removes the notification entries (retryCGlob) and then the transaction code is replaced by the original expression (retryEnd).

4 Correctness of the Concurrent Implementation

In this section we show that *CSHF* can be used as a correct evaluator for *SHF* and its semantics. Hence, we provide a translation from *SHF* into *CSHF*:

Definition 4.1. *The translation ψ of an SHF-process into an CSHF-process is defined homomorphically on the structure of processes: Usually it is the identity on the constructs; the only exception is $\psi(x\,\mathbf{t}\,e) := x\,\mathbf{tg}\,e\,\emptyset\,[]$, i.e. initially, the list of threads to be notified is empty and the TVar is not locked. CSHF-processes $\psi(P)$ where P' is an SHF-process are the* initial *CSHF-processes.*

Since only transactions can introduce local TVars which are removed at the end of a transaction, the following lemma holds:

Lemma 4.2. *Every initial CSHF-process is well-formed provided the corresponding SHF-process is well-formed. Also, every reduction descendant of an initial CSHF-process is well-formed.*

We are mainly interested in *CSHF*-reductions that start with initial *CSHF*-processes. We will show that the translation ψ is adequate. Since ψ is compositional, the hard part is to show that may- and should-convergence are the same for the big-step semantics and the concurrent implementation.

In order to be on solid ground, we first have to analyze the invariants during transactions and properties of the valid configurations.

Lemma 4.3. *The following properties hold during a CSHF-reduction on a well-formed CSHF-process that is reachable from a non-transactional CSHF-process.*

1. *For every component $x\,\mathbf{tl}\,e$, either $x \in L_n$ of the top entry in L, or there is a global TVar x. For every pair of thread identifier u, and TVar-name x, every stack element of the TVar-stack for u contains at most one entry $x\,\mathbf{tl}\,e$.*
2. *If u is in a notify-list of a global TVar, then thread u is transactional.*
3. *Every transaction that starts the commit-phase for thread u by performing the rule (writeStart) is able to perform all other rules until (writeEnd) is performed, without retry, nontermination or getting stuck.*

Also, it is easy to extend the monomorphic type system to *CSHF* and to see that reduction keeps well-typedness.

Definition 4.4. *A CSHF-process P is* successful*, iff it is well-formed and the main thread is of the form* $\langle u \wr y \rangle \overset{\mathsf{main}}{\Longleftarrow}$ `return` e. May- and should-convergence *in CSHF are defined by: $P{\downarrow}_{CSHF}$ iff P is well-formed and $\exists P' : P \xrightarrow{CSHF,*} P' \wedge P'$ successful. $P{\Downarrow}_{CSHF}$, iff P is well-formed and $\forall P' : P \xrightarrow{CSHF,*} P' \implies P'{\downarrow}_{CSHF}$.* Must- and may-divergence *of process P are the negations of may- and should-convergence and are denoted by $P{\Uparrow}_{CSHF}$, and $P{\uparrow}_{CSHF}$, resp., where $P{\uparrow}_{CSHF}$ is also equivalent to $P \xrightarrow{CSHF,*} P'$ such that $P'{\Uparrow}_{CSHF}$.*

Contextual approximation $\leq_{CSHF}$ *and* equivalence $\sim_{CSHF}$ *in CSHF are defined as $\leq_{CSHF} := \leq_{\downarrow_{CSHF}} \cap \leq_{\Downarrow_{CSHF}}$ and $\sim_{CSHF} := \leq_{CSHF} \cap \geq_{CSHF}$ where for $\zeta \in \{\downarrow_{CSHF}, \Downarrow_{CSHF}\}$: $P_1 \leq_{CSHF,\zeta} P_2$ iff $\forall \mathbb{D} \in PC_C : \mathbb{D}[P_1]\zeta \implies \mathbb{D}[P_2]\zeta$.*

Definition 4.5. *If $P_1{\downarrow}_{CSHF} \iff P_2{\downarrow}_{CSHF}$ and $P_1{\Downarrow}_{CSHF} \iff P_2{\Downarrow}_{CSHF}$ then we write $P_1 \sim_{ce} P_2$, A program transformation τ (i.e. a binary relation over CSHF-processes) is* convergence equivalent *iff $P_1\ \tau\ P_2$ always implies $P_1 \sim_{ce} P_2$.*

The reasoning in the following only concerns convergence equivalence, where we have to show: preservation of may-convergence ($P{\downarrow} \Rightarrow \psi(P){\downarrow}_{CSHF}$), reflection of may-convergence ($\psi(P){\downarrow}_{CSHF} \Rightarrow P{\downarrow}$), preservation of should-convergence ($P{\Downarrow} \Rightarrow \psi(P){\Downarrow}_{CSHF}$), and reflection of should-convergence ($\psi(P){\Downarrow}_{CSHF} \Rightarrow P{\Downarrow}$).

Preservation of May-Convergence. We have to show that $P{\downarrow}$ implies $\psi(P){\downarrow}_{CSHF}$, which is not completely straightforward, since the big-step reduction $\xrightarrow{SHFA,*}$ can be tried for free, and only in the case of success, i.e., a $\texttt{return}_{\texttt{STM}}$ is obtained by an atomic transaction, the changes of the $\xrightarrow{SHFA,*}$-reduction sequence are kept. Also, if an `orElse`-expression is reduced, the big-step reduction

(cpBE) $x = \mathbb{E}[x_1] \mid x_1 = x_2 \mid \ldots \mid x_{m-1} = x_m \mid x_m = v$
$\rightarrow x = \mathbb{E}[v] \mid x_1 = x_2 \mid \ldots \mid x_{m-1} = x_m \mid x_m = v$,
if v is an abstraction, a cx-value or a component-name, and $\mathbb{E} \neq [\cdot]$.

(absB) $x = c\ e_1 \ldots e_n \ \rightarrow \ \nu x_1, \ldots x_n.x = c\ x_1 \ldots x_n \mid x_1 = e_1 \mid \ldots \mid x_n = e_n$.

(funrB) Every functional rule (without the surrounding $\mathbb{L}$-context) in a context $x = \mathbb{E}[\cdot]$, but not (cp) and not (abs).

(absG) $x\,\mathbf{tg}\,e\,o\,g \ \longrightarrow\ \nu z.x\,\mathbf{tg}\,z\,o\,g \mid z = e$ where $o \in \{\emptyset, u\}$

(absAt) $\langle x \wr u \rangle \Leftarrow \mathbb{M}_{\texttt{IO}}[(\texttt{atomically}\ e)]] \longrightarrow \langle x \wr u \rangle \Leftarrow \mathbb{M}_{\texttt{IO}}[(\texttt{atomically}\ z)] \mid z = e$.

(gc) $\nu x_1, \ldots, x_n.P \mid x_1 = e_1 \mid \ldots \mid x_n = e_n \ \longrightarrow\ \nu x_1, \ldots x_n.P$
if for all $i = 1, \ldots, n$: x_i does not occur free in P.

Fig. 6. Special Transformations for *CSHF*

is permitted to evaluate the second expression, if it is known that the first one would end in a retry. This is different in *CSHF*, since the execution has to first evaluate the left expression of an `orElse`-expression, and only in case it "retries", the second expression will be evaluated where the changes of the TVars are not kept, but changes belonging to functional evaluation in the bindings are kept. Analyzing the behavior exhibits that these changes of the process can be proved as convergence equivalent transformations.

As a base case, the following lemma holds:

Lemma 4.6. *If P is successful, then $\psi(P)$ is successful.*

An easy case are non-(atomic)-standard reductions:

Lemma 4.7. *If $P_1 \xrightarrow{SHF,a} P_2$ where $a \neq$ (atomic), then $\psi(P_1) \xrightarrow{CSHF} \psi(P_2)$.*

The more complex cases arise for the transactions in the big-step reduction. In this case the state of the concurrent implementation consists of stacks, global TVars and stacks of local TVars. We consider several program transformations related to reduction rules, which are chosen such that it is sufficient to simulate the modifications in the bindings of the retried *CSHF*-standard reductions and then to rearrange reduction sequences of the concurrent implementation.

Definition 4.8 (*CSHF* special transformations). *The* special transformations *are defined in Fig. 6 where we assume that they are closed w.r.t. $\mathbb{D}$-contexts and structural congruence, i.e. for any transformation $\xrightarrow{a}$ with $a \in \{(cpBE), (absB), (funrB), (absG), (absAt), (gc)\}$ we extend its definition as follows: If $P \equiv D[P']$, $Q \equiv D[Q']$, and $P' \xrightarrow{a} Q'$, then also $P \xrightarrow{a} Q$.*

In the appendix (Lemma B.2) we show:

Lemma 4.9. *The rules in Fig. 6 are convergence equivalent.*

First we observe the effects of global retries:

Lemma 4.10. *If there is a CSHF-reduction sequence* $P_1 \xrightarrow{CSHF,*} P_2$*, where an STM-transaction is started in the first reduction for thread u, and the last reduction is a global retry, i.e. (retryEnd) for thread u, of the transaction, then* $P_1 \xrightarrow{*} P_2$ *using the CSHF-special-transformations (cpBE), (absB), (funrB), (absG), and inverse (gc)-transformations from Definition 4.8.*

Proof. A global retry removes all generated local TVars for this thread. There may remain changes in the global TVars and in the bindings: Transformations (absG) may be necessary for global TVars. Functional transformations in the sharing part are still there, but may be turned into non-standard reductions, if these were triggered only by the thread u. Since (cp)-effects in the thread expression are eliminated after a retry in an `orElse`: These may be (cpBE), (absB), (funrB). Various reductions generate bindings which are no longer used after removal of the first expression in an `orElse`, which can be simulated by a reverse (gc).

Lemma 4.11. *If* $P_1 \xrightarrow{SHF,atomic} P_2$*, then* $\psi(P_1) \xrightarrow{CSHF,*} P_2'$*, and* $P_2' \xrightarrow{trans,*} \psi(P_2)$*, where* $\xrightarrow{trans}$ *consists of (cpBE), (absB), (funrB), (absG), and inverse (gc)-transformations in Fig. 6.*

Proof. The correspondence between the contents of the global TVar x in *CSHF* for a single thread u is the local TVar x on the top of the stack, or the global TVar if there is no local TVar for x. The rules that change the bindings permanently in the atomic-transaction reduction are: (atomic), (readl), (readg), (orElse), which can be simulated by sequences of (absG) and reverse (gc). The effects of the functional rules ((cp$_Q$), (abs$_Q$), (mkb$_Q$), (lbeta$_Q$), (case$_Q$)) that survive a retry are either simulated by (mkb$_Q$), (lbeta$_Q$), (case$_Q$) in a binding, or by (cpBE), (absB), or inverse (gc). An (ortry)-reduction in *SHF* can be simulated in *CSHF* by a sequence of reductions starting with (orElse) and ending with (orRetry). However, since (ortry) undoes all changes, in *CSHF* it is necessary to undo the changes in bindings by the special transformations.

Theorem 4.12. *For all SHF-processes P, we have* $P{\downarrow} \implies \psi(P){\downarrow}_{CSHF}$.

Proof. This follows by an induction on the length of the given reduction sequence for P. The base case is covered in Lemma 4.6. Lemmas 4.7 and 4.11 show that if $P_1 \xrightarrow{SHF} P_2$, then there is some *CSHF*-process P_2' such that $\psi(P_1) \xrightarrow{CSHF,*} P_2'$ with $P_2' \sim_{ce} \psi(P_2)$. This is sufficient for the induction step.

Reflection of May-Convergence In this section we distinguish the different reduction steps within a reduction sequence $\psi(P_1) \xrightarrow{*} P_2$ for the different threads u. A (sendRetry)-reduction belongs to the sending thread. A subsequence of the reduction sequence starting with (atomic) and ending with (writeEnd), which includes exactly the u-reduction steps in between, and where no other (atomic) or (writeEnd)-reductions are contained is called a *transaction.* A prefix of a transaction is also called a *partial transaction.* The subsequence of an u-transaction for thread u starting with (writeStart) and ending with (writeEnd) is called the *commit-phase* of the transaction. If the subsequence for thread u starts with (atomic) and ends with (retryEnd), without intermediate (atomic) or (retryEnd), then it is an *aborted transaction*. The subsequence from the first (retryCGlob) until (retryEnd) is the *abort-phase* of the aborted transaction. A prefix of an aborted transaction is also called a *partial aborted transaction*.

Theorem 4.13. *For all SHF-processes P, we have $\psi(P){\downarrow_{CSHF}} \implies P{\downarrow}$.*

Proof. Let $\psi(P_1) \xrightarrow{CSHF,*} P_2$ where P_2 is successful. Since the transactions in the reduction sequence may be interleaved with other transactions and reduction steps, we have to rearrange the reduction sequence in order to be able to retranslate it into *SHF*.
Partial Transactions that do not contain a (writeStart) within the reduction sequence can be eliminated and thereby replaced by interspersed special transformations using Lemma 4.10, which again can be eliminated from the successful reduction sequence by Lemma B.2. If the partial transaction contains a (writeStart), then the missing reduction steps can be added within the reduction sequence before a successful process is reached, since the commit-phase does not change the successful-property of processes.
Aborted Transactions can be omitted since they are replaceable by special transformations, which again can be removed.
Grouping Transactions: We can now assume that within the reduction sequence $\psi(P_1) \xrightarrow{CSHF,*} P_2$ where P_2 is successful, all transactions are completed, and that there are no aborted ones. Now we rearrange the reduction sequence: The (writeEnd)-reduction step is assumed to be the point of attraction for every transaction. Moving starts from the rightmost non-grouped transaction. For this U-transaction, we move the reduction steps that belong to it in the direction of its (writeEnd), i.e., to the right. Non-functional reduction steps belonging to u belong only to u, and can be moved to the right until they are at their place within the transaction, where they may also be commuted with a functional u-reductions if this is triggered by another thread. Functional reduction steps may be triggered by several threads, i.e. may belong to several transactions, but are also moved to the right, where reduction steps can only be commuted if they do not belong to the same thread, The transaction for thread u is complete, if the reduction step (atomic) is moved to its place. Now the reduction sequence is like $Q_1; \xrightarrow{\text{(writeEnd)}}; Q_2; \xrightarrow{\text{(atomic)}}; Q_3$, where Q_1 is the ungrouped part, Q_2 consists only of functional reductions and of `IO`-reductions, and the sequence $\xrightarrow{\text{(atomic)}}; Q_3$ is the part of the reduction where all transactions are already grouped together. Now the reduction sequence is almost in a form that can be backtranslated into *SHF*. The last problem are the local retries in `orElse`-expressions in *CSHF*, which in *SHF* are without effect in the bindings. Using Lemma B.2 it is not hard to see that the (orElse)-reductions, including the retries in the transaction can be replaced by applications of the reduction (orElseND):

(orElseND) $\langle u \wr y\rangle \overset{T,L}{\Longleftarrow} \mathbb{M}_A[\texttt{orElse}\ e_1\ e_2] \mid (u\, \textbf{tls}\, (r : s))$
$\xrightarrow{CSHF} \langle u \wr y\rangle \overset{T,L}{\Longleftarrow} \mathbb{M}_A[e_i] \mid (u\, \textbf{tls}\, (r : s))$ where $i \in \{1, 2\}$

This produces a converging reduction sequence, of standard reductions and (orElseND)-reductions, which, of course, requires to change the exact form of bindings using the special transformations. Looking into the orelse-tree of a single transaction, the obtained reduction sequence for every transaction is the left-most possibility to have a successful transaction. This corresponds exactly to the execution in the big-step semantics. Now we retranslate the reduction sequence including the (orElseND) into an *SHF*-reduction, where the non-transactional reduction steps are exactly translated and the steps within $\xrightarrow{SHFA,*}$ are conformant with the successful path in the `orElse`-tree on the *CSHF*-side. □

Corollary 4.14. *Let P be an SHF-process. Then $P{\Uparrow} \iff \psi(P){\Uparrow}_{CSHF}$.*

Reflection of Should-Convergence is shown as preservation of may-divergence, similar as for the preservation of may-convergence in Theorem 4.12, where now, however, the reduction sequence ends in a must-divergent process. Using Corollary 4.14 as a base case shows that may-divergence is preserved:

Theorem 4.15. *For all SHF-processes P, we have $P{\uparrow} \implies \psi(P){\uparrow}_{CSHF}$.*

Preservation of Should-Convergence This is the last part of the analysis, where we prove the following theorem:

Theorem 4.16. *For all SHF-processes P, we have $\psi(P){\uparrow}_{CSHF} \implies P{\uparrow}$.*

Proof. Assume given a non-transactional *CSHF*-process $\psi(P)$ with $\psi(P) \xrightarrow{CSHF,*} P_1$ and $P_1{\Uparrow}_{CSHF}$. The reasoning is as in Theorem 4.13 with some differences, since we have to ensure the condition $P_1{\Uparrow}_{CSHF}$, which is more complex than the "successful"-condition. *Partial transactions:* If it includes a (writeStart) or a (retryCGlob), then the transaction can be completed by $P_1 \xrightarrow{CSHF,*} P_1'$ where $P_1'{\Uparrow}_{CSHF}$, and so we can assume that these do not exist. If the partial transaction does neither include a (writeStart) nor a (retryCGlob), then using the same arguments as in Theorem 4.13, we see that we can assume that the partial transaction is grouped before P_1, i.e. the transaction ends with the partial transaction $P_1'' \xrightarrow{CSHF,*} P_1$. Lemma C.1 shows that $P_1''{\uparrow}_{CSHF}$, which permits to assume that the partial transaction can be omitted. *Aborted transactions:* can be omitted since they are replaceable by special transformations, which again can be removed due to Lemma B.2. *Grouping transaction:* same arguments as in the proof of Theorem 4.13. *Final retranslation:* similar to the proof of Theorem 4.13.

Summary. We have proved in Theorems 4.12, 4.13, 4.15, and 4.16 that the translation ψ mapping *SHF*-processes into *CSHF*-processes is convergence equivalent. Together with compositionality of ψ, we obtain observational correctness and adequacy:

Main Theorem 4.17. The translation $\psi : \mathit{SHF} \to \mathit{CSHF}$ is observational correct, i.e. for all process contexts D and *SHF*-processes P the equivalences $D[P]{\downarrow} \iff \psi(D)(\psi(P)){\downarrow}_{\mathit{CSHF}}$ and $D[P]{\Downarrow} \iff \psi(D)(\psi(P)){\Downarrow}_{\mathit{CSHF}}$ holds. This also implies that ψ is adequate, i.e. $\psi(P_1) \sim_{\mathit{CSHF}} \psi(P_2) \implies P_1 \sim_c P_2$.

This shows that *CSHF* is a correct evaluator for *SHF*-processes w.r.t. the big-step semantics.

Instead of permitting to retry a transaction too often, the strategy from [6] can be applied: activate retried transaction Q only if some of the read TVars is modified. The proofs above can be used to show that this restriction of reductions is equivalent to the *CSHF*-semantics.

Note that omitting the send retry for abortion would make the implementation incorrect (non-adequate): a thread that runs into a loop if TVar x contains a 1 and returns otherwise, may block this thread indefinitely, which is not the case in the specification *SHF*.

5 Related Work

General remarks on STM can be found in [3,4]. [3] argue that more research is needed to reduce the runtime overhead of STM. We believe that ease of maintenance of programs and the increased concurrency provided by STM may become more important in the future. The paper which strongly influenced our work is [6]. *SHF* borrows from the operational semantics of STM Haskell, where differences are that our calculus is extended by futures, models also the call-by-need evaluation, but does not include exceptions and is restricted to monomorphic typing. [6] describe the current implementation of STM Haskell in the Glasgow Haskell Compiler (GHC), however no formal treatment of this implementation is given. The approach taken in the GHC implementation is close to ours with the difference that instead of aborting transactions by the committing transaction (i.e. sending retries in *CSHF*), transactions abort and restart themselves by temporarily checking the local transaction log against the status of the global memory, and thus detecting conflicts. This is comparable to our approach, and we are convinced that the implementations are closely related. However, modelling their approach in a formal semantics would require more effort due to pointer-equality, for example. A potential semantical problem in [6] is that exceptions may make local values of TVars visible outside the transaction.

A semantical investigation of STM is in [2], where a call-by-name functional core language with concurrent processes is defined, and a contextual equivalence is used as equality. Also strong results are obtained by proving correctness of the monad laws and other program equivalences w.r.t. their semantics. However, [2] only considers may-convergence in the contextual equivalence, which is too weak for reasoning about non-deterministic (and concurrent) calculi. Also, `seq` is missing in [2] which is used in Haskell and known to change the semantics, such that validity of the monad laws only holds under further typing restrictions (see [10]). A further difference to our work is that [2] use pointer equality.

[7] propose to investigate correctness of an implementation, but stick to testing. [1] consider correctness of implementing STM in a small calculus with call-by-value reduction and a monadic extension similar to the STM/IO-extension in [6]. The main reasoning tool is looking for traces of effects, and arguing about commuting and shifting the effects within traces, where several important properties are proved It is hard to compare the results with ours, however, from an abstract level, their proof method appears to ignore the should-convergence restriction: There is no argument on forced aborts of transactions.

6 Conclusion

We have presented a big-step semantics for STM-Haskell as a specification, and a small-step concurrent implementation. Using formal reasoning and the strong notion of contextual equivalence with may- and should-convergence we prove correctness of the implementation. As a proof of concept we implemented a prototype of our approach in Haskell[1], which gives evidence of the correct design of *CSHF*.

Further research directions are to consider smarter strategies for earlier aborts and retries of conflicting transactions, extending the language, for example by exceptions, and a polymorphic type system.

References

1. Bieniusa, A., Thiemann, P.: Proving isolation properties for software transactional memory. In: Proc. ESOP'11. Volume 6602 of LNCS. (2011) 38–56
2. Borgström, J., Bhargavan, K., Gordon, A.D.: A compositional theory for STM Haskell. In: Proc. Haskell '09, ACM (2009) 69–80
3. Cascaval, C., Blundell, C., Michael, M.M., Cain, H.W., Wu, P., Chiras, S., Chatterjee, S.: Software transactional memory: why is it only a research toy? Commun. ACM **51**(11) (2008) 40–46
4. Harris, T., Larus, J.R., Rajwar, R.: Transactional Memory, 2nd edition. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers (2010)
5. Harris, T., Marlow, S., Peyton Jones, S., Herlihy, M.: Composable memory transactions. In: Proc. PPoPP'05, ACM (2005) 48–60
6. Harris, T., Marlow, S., Peyton Jones, S.L., Herlihy, M.: Composable memory transactions. Commun. ACM **51**(8) (2008) 91–100
7. Hu, L., Hutton, G.: Towards a verified implementation of software transactional memory. In: Proc. TFP'08. Volume 9., Intellect (2009) 129–144
8. Rensink, A., Vogler, W.: Fair testing. Inform. and Comput. **205**(2) (2007) 125–198
9. Sabel, D., Schmidt-Schauß, M.: A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. Math. Structures Comput. Sci. **18**(03) (2008) 501–553
10. Sabel, D., Schmidt-Schauß, M.: A contextual semantics for Concurrent Haskell with futures. In: Proc. PPDP'11, ACM (2011) 101–112

[1] The source code is available from http://www.ki.cs.uni-frankfurt.de/research/stm

11. Sabel, D., Schmidt-Schauß, M.: Conservative concurrency in Haskell. In: Proc. LICS'12, IEEE (2012) 561–570
12. Schmidt-Schauß, M., Niehren, J., Schwinghammer, J., Sabel, D.: Adequacy of compositional translations for observational semantics. In: Proc. IFIP TCS'08. Volume 273 of IFIP., Springer (2008) 521–535
13. Shavit, N., Touitou, D.: Software transactional memory. In: Proc. PODC'95, ACM (1995) 204–213
14. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing, Special Issue **10** (1997) 99–116
15. van Eekelen, M.C.J.D., Plasmeijer, M.J., Smetsers, J.E.W.: Parallel graph rewriting on loosely coupled machine architectures. In: Proc. CTRS'90. Volume 516 of LNCS., Springer (1990) 354–369

A Typing Rules for *SHF*

$\texttt{types}(c)$ is the set of monomorphic types of constructor or monadic operator c

$$\frac{\Gamma(x)=\tau,\quad \Gamma\vdash e::\texttt{IO}\ \tau}{\Gamma\vdash \langle u \wr x\rangle \Leftarrow e::\texttt{wt}}\qquad \frac{\Gamma(x)=\tau,\quad \Gamma\vdash e::\tau}{\Gamma\vdash x=e::\texttt{wt}}\qquad \frac{\Gamma\vdash P_1::\texttt{wt},\quad \Gamma\vdash P_2::\texttt{wt}}{\Gamma\vdash P_1 \mid P_2::\texttt{wt}}$$

$$\frac{\Gamma(x)=\texttt{TVar}\ \tau,\quad \Gamma\vdash e::\tau}{\Gamma\vdash x\,\mathbf{t}\,e::\texttt{wt}}\qquad \frac{\Gamma\vdash P::\texttt{wt}}{\Gamma\vdash \nu x.P::\texttt{wt}}\qquad \frac{\Gamma(x)=\tau}{\Gamma\vdash x::\tau}$$

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

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

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

$$\frac{\forall i:\Gamma\vdash e_i::\tau_i,\quad \tau_1\to\ldots\to\tau_n\to\tau_{n+1}\in\texttt{types(c)}}{\Gamma\vdash(c\ e_1\ \ldots\ e_{\mathrm{ar}(c)})::\tau_{n+1}}\quad \begin{array}{l}\text{where } c \text{ is a constructor,}\\ \text{or a monadic operator}\end{array}$$

Fig. 7. Typing rules

B Convergence Equivalence of Special Transformations

In this section we show convergence-equivalence of special transformations in the concurrent calculus *CSHF*. We also prove convergence-equivalence of the further rule (cpxgc), shown in Fig. 8, which is necessary to close critical overlappings. We also assume that (cpxgc) is closed w.r.t. $\equiv$ and PC_C-contexts.

Since for the rules of the calculus *CSHF*, the variable (binding-) chains are transparent, there is no need to copy binding-variables to other places in the expressions, which strongly reduces the number of critical overlappings. This, however, enforced that (abs) is in the set of standard reductions.

In the following we use forking and commuting diagrams, where in general we only write the forking diagrams. A forking diagram shows (in an abstract way, since processes are omitted) how a given overlapping $\xleftarrow{CSHF}\xrightarrow{T}$ (the fork) between a standard reduction and a transformation T can be closed, a commuting diagram shows how a sequence $\xrightarrow{T}\xrightarrow{SHF}$ can be closed, i.e. how a transformation and a reduction can be commuted. A set of diagrams for a transformation T is complete if for every concrete fork $P_1 \xleftarrow{CSHF} P_2 \xrightarrow{T} P_3$ (or sequence $P_1 \xrightarrow{T} P_2 \xrightarrow{SHF} P_3$) at least one diagram of the set is applicable, which means that the corresponding transformations and reductions exist (on the concrete level). Since there may be several concurrent threads, the diagrams have to express reduction commutations of concurrent reductions. In order to have simple

(cpxgc) $\nu x.C[x,\ldots,x]] \mid x = y \;\longrightarrow\; C[y,\ldots,y]]$
where C is a context, where all holes are in an $\mathbb{A}$-context, and all occurrences of x are indicated in the notation, i.e., there are no other occurrences.
$\mathbb{A}$ is the class of expression contexts where the hole is not within an abstraction nor in a letrec-expression, nor in an alternative of a case and not in an argument of a constructor.

Fig. 8. The (cpxgc) transformation

$$\text{(trans)}\quad\begin{tikzcd}\cdot \arrow[r, "trans"] \arrow[d, "{CSHF,a}"'] & \cdot \arrow[d, dashed, "{CSHF,a}"] \\ \cdot \arrow[r, dashed, "trans"'] & \cdot\end{tikzcd}\qquad\begin{tikzcd}\cdot \arrow[r, "trans"] \arrow[d, "{CSHF,a}"'] & \cdot \arrow[dl, dashed, "{CSHF,a}"] \\ \cdot & \end{tikzcd}\qquad\begin{tikzcd}\cdot \arrow[r, "trans"] \arrow[d, "{CSHF,a}"'] & \cdot \arrow[dl, dashed, equal] \\ \cdot & \end{tikzcd}$$

Fig. 9. The standard: square and triangle diagrams of special transformations in Fig. 6 and 8

$$\text{(absG)}\quad\begin{tikzcd}\cdot \arrow[r, "absG"] \arrow[d, "{CSHF,a}"'] & \cdot \arrow[d, dashed, "{CSHF,a}"] \\ \cdot & \cdot \arrow[l, dashed, "gc"]\end{tikzcd}\qquad\text{(absAt)}\quad\begin{tikzcd}\cdot \arrow[r, "absAt"] \arrow[d, "{CSHF,a}"'] & \cdot \arrow[d, dashed, "{CSHF,a}"] \\ \cdot & \cdot \arrow[l, dashed, "cpxgc"]\end{tikzcd}$$

$$\text{(absB)}\quad\begin{tikzcd}\cdot \arrow[r, "absB"] \arrow[d, "{CSHF,a}"'] & \cdot \arrow[d, dashed, "{CSHF,a}"] \\ \cdot & \cdot \arrow[l, dashed, "cpxgc"]\end{tikzcd}\qquad\text{(cpxgc)}\quad\begin{tikzcd}\cdot \arrow[r, "cpxgc"] \arrow[d, "{CSHF,a}"'] & \cdot \arrow[d, dashed, "{CSHF,a}"] \\ \cdot \arrow[r, dashed, "gc"'] & \cdot\end{tikzcd}$$

Fig. 10. The unusual forking diagrams of special transformations in Fig. 6 and 8

forms of diagrams, at most a single sr-reduction should be vertically in the diagrams on the west- and east-side. If more standard reductions would be necessary, then reasoning is far more complex.

Lemma B.1. *The forking diagrams of the transformations in Fig. 6 and 8 with standard-reductions may be square or triangle diagrams or the equality (see Fig. 9). The unusual forking diagrams of the transformations in Fig. 6 and 8 are in Fig. 10. The commuting diagrams can be obtained from them by taking the same diagram where the existential and given vertical arrows are switched. The equality diagram may only occur as follows: an (absB)-transformation may be a (CSHF,abs)-reduction, a (cpBE)-transformation may be a (CSHF,cp)-reduction, and a (funrB)-transformation may be a functional standard reduction.*

Proof. We provide arguments for every reduction, and also exhibit exceptional diagrams in Fig. 11. For a (gc) transformation it is easy to verify that bindings cannot be removed by any standard reduction, and standard reductions do not interfere with the removed bindings by (gc). For (cpBE) the restriction of the target positions shows that there is no fork, where the to-be-copied expression of a standard reduction is modified by (cpBE). The sharing mechanism in, for example, (orElse) and (readg) shows that there is no duplicate of (cpBE) in the

Forking with a (atomic) standard reduction:

$$\begin{tikzcd}
\langle u \wr x\rangle \Leftarrow \mathtt{atomically}\ e \arrow[r, "absAt"] \arrow[d, "{CSHF,(atomic)}"'] & \langle u \wr x\rangle \Leftarrow \mathtt{atomically}\ z \mid z = e \arrow[d, dashed, "{CSHF,(atomic)}"] \\
\langle u \wr x\rangle \Leftarrow \mathtt{atomically!}\ z\ z \mid z = e & \langle u \wr x\rangle \Leftarrow \mathtt{atomically!}\ z'\ z' \mid z' = z \mid z = e \arrow[l, dashed, "cpxgc"]
\end{tikzcd}$$

Forking standard reduction (cp) with (cpxgc):

$$\begin{tikzcd}
\nu x.E[x] \mid x = y \mid y = v \arrow[r, "cpxgc"] \arrow[d, "{CSHF,cp}"'] & \nu x.E[y] \mid y = v \arrow[d, dashed, "{CSHF,cp}"'] \\
\nu x.E[v] \mid x = y \mid y = v \arrow[r, dashed, "gc"] & \nu x.E[v] \mid y = v
\end{tikzcd}$$

Forking (writeTV) with (absG)

$$\begin{tikzcd}
x\,\mathbf{tg}\,e\,o\,g\ldots \arrow[r, "absG"] \arrow[d, "{CSHF,writeTV}"'] & x\,\mathbf{tg}\,z\,o\,g \mid z = e \arrow[d, dashed, "{CSHF,writeTV}"] \\
x\,\mathbf{tg}\,e'\,o\,g\ldots & x\,\mathbf{tg}\,e'\,o\,g \mid z = e \arrow[l, dashed, "gc"']
\end{tikzcd}$$

Forking (cp) with (absB)

$$\begin{tikzcd}
E[x] \mid x = c\ y_1\ y_2\ldots \arrow[r, "absB"] \arrow[d, "{CSHF,cp}"'] & E[x] \mid x = c\ z_1\ z_2 \mid z_1 = y_1 \mid z_2 = y_2 \arrow[d, dashed, "{CSHF,cp}"] \\
E[c\ y_1\ y_2] \mid x = c\ y_1\ y_2\ldots & E[c\ z_1\ z_2] \mid x = c\ z_1\ z_2 \mid z_1 = y_1 \mid z_2 = y_2 \arrow[l, dashed, "cpxgc"']
\end{tikzcd}$$

Fig. 11. Examples for diagrams in Lemma B.1

south edge of diagrams. A further argument is that the standard reduction is deterministic within threads. For (funrB) the reduction takes place inside a binding an never inside a thread, hence standard reductions and (funrB) only have trivial overlappings. The (absG) transformation may critically overlap with a (writeTV) standard reduction. In this case a (gc) is required to remove the created bindings. The (absAt) transformation can critically overlap with the (atomic) reduction where (cpxgc) is necessary to remove indirections. The transformation (absB) may critically overlap with a (cp) that copies a cx-value where again (cpxgc) can be used to remove indirections. The exceptional diagram for (cpxgc) using (gc) in the south arrow occurs for example in a forking of (case), (seq), or (lbeta) with (sendRetry).

Lemma B.2. *The rules in Fig. 6 and 8 are convergence equivalent for may-convergence and should-convergence in the calculus CSHF.*

Proof. We first consider may-convergence. Note that due to the reverse transformation steps in the diagrams, it is necessary to show that the transformations preserve and reflect convergence in one induction. Let $P' \xleftarrow{CSHF,k} P \xrightarrow{\tau} P''$ where τ is any of the special transformations or their inverse transformation, P'

is successful and $k \geq 0$. By induction on k we show that P'' is may-convergent. For the base case, P is already successful. Then P'' must be successful, too, which follows by inspection of the definitions of the transformations. For the induction step assume $P' \xleftarrow{CSHF,k} P_1 \xleftarrow{CSHF} P \xrightarrow{\tau} P''$ for some $k \geq 0$. Then depending on the direction of τ we apply a forking or a commuting diagram of Lemma B.1 to $P_1 \xleftarrow{CSHF} P \xrightarrow{\tau} P''$. Then there are the following cases:

- If the second or the third diagram of Fig. 9 is applied, then either $P'' \xrightarrow{CSHF} P_1$ or $P'' = P_1$. In both cases this implies $P''{\downarrow}_{CSHF}$.
- If any other diagram is applied, then there exists a process P_2 and a special transformation or its inverse τ' such that $P_1 \xrightarrow{\tau'} P'' \xrightarrow{CSHF} P_2$. Now we apply the induction hypothesis to $P' \xleftarrow{CSHF,k} P_1 \xrightarrow{\tau'} P_2$ and derive $P_2{\downarrow}_{CSHF}$. Since $P'' \xrightarrow{CSHF} P_2$, we also have $P''{\downarrow}_{CSHF}$.

The proof for should-convergence is completely analogous except for the base case: The base case requires that if $P \xrightarrow{\tau} P'$ then $P{\Uparrow}_{CSHF} \iff P'{\Uparrow}_{CSHF}$. But this holds by equivalence w.r.t. may-convergence.

C Should-Convergence Preservation

The following property is required in the proof of preservation of should-convergence (reflection of may-divergence): That partial transactions can be eliminated:

Lemma C.1. *Let P_0 be a nontransactional process, $P_0 \xrightarrow{CSHF,*} P_1$ with $P_1{\Uparrow}_{CSHF}$, such that $P \xrightarrow{Q} P_1$ is a suffix of the reduction and $P \xrightarrow{Q} P_1$ is a partial transaction for a thread u starting with (atomic) but the commit-phase and the retry-phase is missing, i.e. there is no reduction (writeStart) nor a (retryCGlob) for thread u. Then $P{\Uparrow}_{CSHF}$.*

Proof. The proof is by contradiction. Assume that $P{\downarrow}_{CSHF}$, i.e. $P \xrightarrow{CSHF,*} P_2$, where P_2 is successful. Then we have to argue that this implies $P_1{\downarrow}_{CSHF}$.

$$\begin{tikzcd}
P \arrow[r, "Q{,}u", "CSHF"'] \arrow[d] & P_1{\Uparrow} \arrow[d, dashed, "CSHF"', "?"] \\
P_2 \text{ (success)} & \cdot
\end{tikzcd}$$

The assumption that P_0 is a nontransactional process implies that the reduction sequences are reachable and thus the notifications and memory at the thread arrows and the Tvars are consistent.
Using the claims in the proof of Theorem 4.13, we can assume that in the reduction sequence $P \xrightarrow{CSHF,*} P_2$ all transactions are completed and that there are no aborted transactions.

The goal is now to construct a converging *CSHF*-reduction sequence for P_1. The idea is to use the same reduction steps as for P. Let us ignore the functional and the `IO`-reduction steps and concentrate on the transactional ones. The reduction steps can be shifted from $P \xrightarrow{CSHF,*} P_2$ to a reduction sequence for P_1. Note that this shifting may also be accompanied by modified notification entries of TVars.
There are two cases:

1. If the TVars read by the transaction Q are unchanged in $P \xrightarrow{CSHF,*} P_2$ (up to bindings via variable-chains), then a construction of a reduction of P_1 to a successful process is possible in a standard way, where the u-transaction is either unused or will be completed.
2. Let V be the set of global TVars read by the transaction Q. Assume some TVar from V is updated in the prefix of $P \xrightarrow{CSHF,*} P_2$, before any u-transaction starts. then we can again construct a converging standard reduction for P_1: Transporting the reduction steps to P_1, the only difference are the notifications for the TVars read by the u-transaction. There will be an extension by a (sendRetry) that aborts the (image of the) Q-transaction, which we will add immediately after a commit of the updates.
   After the retry the state (i.e. the process) is the same in the two reductions up to some special transformations. Now we can proceed with the construction that transports the reductions from the successful reduction sequence to the P_1-reduction sequence. This will end in a converging standard reduction sequence for P_1.

This contradicts the assumption that P_1 is must-divergent.