Static Application-Level Race Detection in STM Haskell using Contracts

Romain Demeyer
University of Namur, Belgium
romain.demeyer@unamur.be

Wim Vanhoof
University of Namur, Belgium
wim.vanhoof@unamur.be

Writing concurrent programs is a hard task, even when using high-level synchronization primitives such as transactional memories together with a functional language with well-controlled side-effects such as Haskell, because the interferences generated by the processes to each other can occur at different levels and in a very subtle way. The problem occurs when a thread leaves or exposes the shared data in an inconsistent state with respect to the application logic or the real meaning of the data. In this paper, we propose to associate *contracts* to transactions and we define a program transformation that makes it possible to extend *static contract checking* in the context of STM Haskell. As a result, we are able to check statically that each transaction of a STM Haskell program handles the shared data in a such way that a given consistency property, expressed in the form of a user-defined boolean function, is preserved. This ensures that bad interference will not occur during the execution of the concurrent program.

1 Introduction

Software Transactional Memory (STM) [43] is supposed to help us in the complex task of writing concurrent programs. The pure and lazy functional language Haskell proposes a particularly clean and straightforward integration of STM [23, 16, 15] in its concurrent version. Shared variables, called *transactional variables* (*TVar*s) in the context of STM, can be accessed by different threads using the STM primitives `readTVar` and `writeTVar`, and the programmer can protect those accesses from the interference of other threads by making them (conceptually) atomic using the primitive `atomically`. In fact, *TVar*s can *only* be accessed from within such atomic blocks, also called *transactions*. While the use of STM Haskell allows to avoid many tricky low-level bugs, such as low-level race conditions and deadlocks, this in itself is not an absolute guarantee for correctness [23]. Indeed, in spite of STM being a beautiful tool that allows one to get rid of low-level locking mechanisms and to focus on higher-level aspects of the program, it does not prevent all errors related to concurrent programming. In particular, a fundamental difficulty related to concurrent programming with shared variables remains: the risk of exposing those data in an inconsistent state *with respect to the application logic* [4].

Let us illustrate this problem on a simple example. Consider the source code implementing a function `f` presented in the top part of Fig. 1, where the `do` notation refers to the classic syntactic sugar to express monadic computations [16, 18]. We suppose the existence of two *TVar*s: `shSum`, which stores an integer, and `shTab` which stores a list of integers. There are two nested functions in the main function `f`. The first one, `addTab`, consists of two STM operations that allows to update `shTab` by adding an integer `n` in front of the list[1]. The second one, `addSum`, updates the other *TVar*, `shSum`, by adding `n` to its value. When considering this definition, each update is protected by the operation `atomically`, and, consequently, this code is free of low-level race conditions (i.e. concurrent access to the same data element with at

[1] Let `x` be an integer and `xs` be a list of integers $[x_1, ..., x_n]$, `(x:xs)` correspond to the list $[x, x_1, ..., x_n]$.

N. Yoshida, W. Vanderbauwhede (Eds.): Programming Language Approaches to Concurrency- and Communication-cEntric Software 2013 (PLACES'13)
EPTCS 137, 2013, pp. 115–134, doi:10.4204/EPTCS.137.10

least one access being a write). This is both sufficient and efficient, as long as the values of the *TVar*s are independent. However, if there is an implicit link between the values of those variables, the story is more subtle. Suppose that `shSum` is meant to represent at all times the sum of the integers in the list `shTab`. In this case, an inconsistent state (in which the value of `shTab` has been updated while the value of `shSum` has not) is exposed *between* the two transactions, which may be problematic in a concurrent program. Indeed, suppose for example that the thread's execution is (conceptually) suspended at this precise point, while another thread doubles the value of `shSum` and that of each integer of `shTab`. At the end of the execution of both threads, the sum of the integers in the list `shTab` and the value of `shSum` will be different, breaking the programmer's intention and thus showing an unacceptable error in the program. This situation is what is sometimes called an *application-level race condition* [4, 9], as it represents an inconsistency with respect to the *logic* of the application that cannot be observed from the source code alone. In the context of our example, it can be easily corrected by encapsulating both updates in a single transaction, as depicted by the alternative implementation of `f` sketched at the bottom part of Fig. 1.

```
f n = let addTab n = do tab <- readTVar shTab
                        writeTVar shTab (n:tab)
          addSum n = do s <- readTVar shSum
                        writeTVar shSum (s+n)
          in do atomically ( addTab n )
                atomically ( addSum n )
```

```
f n = let ...
          in do atomically ( do addTab n
                                addSum n )
```

Figure 1: The function `f` implemented with two transactions (top part) and with a single transaction (bottom part)

As application-level race conditions appear often and in a more subtle form in large programs [26] and as they are hard to prevent with testing, there is a certain interest in having a tool that is able to detect them statically. This boils down to verifying that each transaction preserves the *TVar*s in a state that is consistent with respect to the given application logic.

One interesting approach towards specification and verification of program properties of Haskell programs is so-called *static contract checking* [47] which has been developed for a core version of the language. Its convenience lies in the fact that the property to be checked can be specified by writing it in the form of a Haskell function, which liberates the programmer from the need of dealing with a separate specification language [47]. Unfortunately, not being designed to handle concurrent programs, the technique does not handle mutable states nor transactions. The goal of this paper is to overcome this limitation.

More specifically, we make the following contributions:

1. We define contract checking for the language used in the transactions of STM Haskell programs. For this purpose, we have defined a novel kind of contract, dedicated to STM operations.
2. We re-express the problem of the detection of application-level race conditions in the context of contract checking.
3. We propose a practical sound method to prove automatically contract satisfaction.

The method we propose in order to achieve that last goal is to transform expressions and contracts in a such form that an existing verification technique, introduced in Section 2, can be used. Our framework is presented in broad terms with the help of a motivating example in Section 3. Then, we define formally the framework, we prove our transformation to be correct and we discuss extensions and limitations in Section 4, discuss how we could overcome some current limitations of our approach, before replacing our results in the context of related work (Section 5).

2 Background: Static Contract Checking for Haskell

The static verification framework of Xu et al.[47] is based on the notion of *contracts* [32, 2]. A contract can be seen as a refinement of the type of a function. For example, let us consider the function `inc` depicted in Fig 2. The type of the function tells us that it takes an integer as argument and returns an integer as well. The contract gives more information about the function by telling us that the integer expected as argument has to be strictly positive, and that the value returned has to be strictly greater than the argument.

```
inc :: Int -> Int                       -- Type
inc :: { x | x > 0 } -> { r | r > x }   -- Contract
inc x = x + 1                           -- Definition
```

Figure 2: The `inc` function, with its type and contract.

In the context of this example, contract *checking* consists then in verifying that if the argument fulfills its part of the contract, then the value returned fulfills its own. In the framework of [47], this checking is done in two steps. First, the contracts are integrated into the function definition in a such way that the function explicitly fails by returning a special value if they are not fulfilled. The function transformed in this way is called the *wrapped function*. Secondly, symbolic execution is used to check whether the wrapped function *can* effectively fail. For our example, the wrapped function is depicted in Fig 3. The outer case expression represents the fact that we assume a strictly positive value for the argument, i.e. the opposite branch is explicitly tagged as unreachable (by returning the special value `UNR`). The inner case expression represents the fact that the function must fail (by returning the special value `BAD`) if the returned value is not greater than the input. In other words, `inc'` behaves just like `inc` except that it returns `UNR` if its argument is negative and `BAD` if the function definition violates the contract, i.e. returns a value that is no strictly greater than the input.

The second step consists in simplifying the wrapped function in order to prove that all `BAD` branches can be safely removed, which is quite easy in this example with simple symbolic execution and basic theorem proving which can replace `x + 1 > x` by `True` [47]. Note that when the function is *called* in the program, the call in question is also replaced by a wrapped call, similar to the wrapped function definition apart from the fact that `BAD` and `UNR` are swapped. This depicts the fact that checking a function definition corresponds to verifying the postcondition, assuming that the precondition holds, while checking a function call corresponds to verifying that the precondition holds and then, if so, assuming that the postconditions holds. This method allows to verify entire programs in a modular way and to deal adequately with recursion [47].

Contract checking of a function being a undecidable problem, it has three possible outcomes: either the function is definitely safe (all `BAD`s are removed during simplification and hence proven to be unreachable), either definitely not safe (the expression does simplify to `BAD`), or unknown (some `BAD`s

```
inc' x = case x > 0 of
           True -> case x + 1 > x of
                     True  -> x+1
                     False -> BAD
           False -> UNR
```

Figure 3: The function that is build based on `inc` and its contracts.

remain present after simplification, but we cannot prove that the expression will actually fail). However, by using a suitable inlining/simplification strategy, a considerable amount of programs can be proven to be correct with respect to their given contracts [47]. As an example, our own prototype implementation based on [47] succeeds in the verification of the somewhat more involved example represented in Fig. 4.

```
add :: Int -> ([Int],Int) -> ([Int],Int)
add :: { x | True} -> { (tab,s) | sum tab == s} -> { (tab,s) | sum tab == s}
add n (tab,s) = (n:tab,s+n)

sum :: [Int] -> Int
sum xs = case xs of []     -> 0
                    (l:ls) -> l + sum ls
```

Figure 4: The `add` function and its type and contract.

3 The Main Idea

In this section, we present in an intuitive fashion a framework that allows to statically detect application-level race conditions in a STM Haskell program. It consists, in other words, in verifying that each transaction preserves the transactional variables (*TVar*s) in a consistent state. The main idea behind the verification is to deduce contracts from the properties expressing consistency of the *TVar*s, and to subsequently verify these contracts by an adaptation of the static contract checking framework in order to make it deal with STM Haskell. The basic difficulty in using the framework of Xu et al. [47] is that the latter is not designed to deal with *mutable* variables and side-effects, i.e. STM and I/O primitives. However, since transactions in STM Haskell do not produce side effects other than updating the values of some *TVar*s, they can be seen as pure functions taking the values of the *TVar*s they manipulate as input, and producing a set of new values for them. To illustrate our approach, we will show how it is capable of detecting an inconsistency in the function of the top part of Fig. 1, while it proves the alternative function (sketched at the bottom part) to be application-level race condition free.

As a first step, one needs to *specify* what it means for the *TVar*s to be in a consistent state. This can be done by writing a Haskell function that returns `True`, respectively `False`, if the set of *TVar*s are in a consistent, resp. inconsistent, state. In the context of our running example, this function would be as follows:

```
inv (shTab,shSum) = sum shTab == shSum
  where sum xs = case xs of []     -> 0
                            (l:ls) -> l + sum ls
```

Indeed, in our program, the *TVar*s are considered to be in a consistent state if the sum of the elements from the list stored in `shTab` equals the integer stored in `shSum`. Note that the above expression is the

only information that needs to be specified for our approach to be capable of verifying the absence of violations of this consistency definition. Moreover, our framework allows this function to be *any* Haskell function that returns a boolean, including functions whose definition involves calls to recursive functions that are defined elsewhere in the program.

From the above function, we generate the following *STM contract*, that we call the *transactional invariant*:

```
INVARIANT :: || c <> c || Any     where c = {(shTab,shSum) | inv(shTab,shSum)}
```

While the language of the contracts will be formally defined further down the paper, intuitively the above contract specifies that if the *TVar*s are in a consistent state at the very beginning of the transaction – i.e. their content satisfies the contract `c` at the left of `<>`, then, it must also be `True` at the very end of the transaction – i.e. their content satisfies the contract `c` at the right of `<>`. Formally, like any Haskell expression, also a transaction in STM Haskell returns a value, but in the example we don't care about it – hence the `Any` in the contract.

To achieve the verification of this contract, we define an operator, which we denote by $\mathscr{T}$, that transforms a STM Haskell expression e, i.e. an expression which involves mutable variables and STM primitives, into a basic non-concurrent Haskell expression $\mathscr{T}(e)$ – i.e. an expression which is completely *pure* – in such a way that the contracts can be checked on $\mathscr{T}(e)$ by the non-concurrent framework of [47], while the results of the analysis are valid for the contracts in the original concurrent program e. The intuitive idea behind the transformation is to represent the effect of a transaction on the *TVar*s by a pure function (a lambda abstraction) that takes as arguments not only the potential free variables of the transaction, but also the values of those *TVar*s as input, and that computes a vector containing the value computed by the transaction *and* the values of the (updated) *TVar*s.

In our example, we transform the three transactions from Fig. 1 into the three following lambda expressions:

```
\n (shTab,shSum) -> ((),(n:shTab,shSum))
\n (shTab,shSum) -> ((),(shTab,shSum+n))
```

```
\n (shTab,shSum) -> ((),(n:shTab,shSum+n))
```

Indeed, for each lambda expressions, the arguments are `n`, which is the only free variable in the transactions, and `(shTab,shSum)`, which is a couple representing, intuitively, the value of the *TVar*s at the beginning of the transaction. The lambda expressions express how *TVar*s are updated with respect to those arguments. We can also transform the transactional invariant into a (pure) function contract which bears no reference to STM:

```
INVARIANT :: Ok -> c -> (Any,c)     where c = {(shTab,shSum) | inv(shTab,shSum)}
```

Intuitively, an expression e satisfies this contract if, when two arguments are applied to e such that the first one does not crash, i.e. hence the `Ok` that will be defined in Section 4, and the second one satisfies c, then it produces a couple of elements such that the second one also satisfies c. Verifying whether this contract is satisfied by the three transformed transactions is then an instance of standard contract checking for pure expressions with respect to pure contracts [47]. The verification will prove failure for the two first transactions and success for the third one. These results are easily transposable to the original concurrent functions as they state that the function at the top part does not preserve the consistency of the *TVar*s (indicating an application-level race conditions) whereas the one at the bottom part does.

4 Our Framework in Detail

In this section, we present a core language based on STM Haskell, denoted $\mathscr{H}$, the language of *contracts*, and we formally develop the $\mathscr{T}$-operator that allows to transform expressions and contracts such that they can be checked by standard contract checking.

4.1 The language

A STM Haskell program can be seen as a series of I/O operations. Among the different kinds of I/O operations (reading/writing a file, creating a thread,...) is the `atomically` operation which allows to perform a series of STM operations in a conceptually atomic way with respect to the other threads. Such a series of STM operations embedded in an `atomically` operation is called a *transaction*. STM operations consist basically in reading and updating transactional variables (*TVar*s), and the *only* way to perform these is from within an `atomically` operation (a fact that is guaranteed by the type system of STM Haskell). Performing I/O operations is *not* allowed inside transactions.

Fig. 5 presents the syntax of $\mathscr{H}$, the core language used for writing a STM Haskell transaction. As such, the language is only a subset of the one defined in [15] and [6] but it allows to define all parts of a STM Haskell program that can be used from within a transaction, including STM operations, lambda abstractions and (recursive) function definitions. Note that $\mathscr{H}$ does not contain an `atomically` primitive, as the latter is an IO operation and IO operations are not permitted within a transaction. Consequently, transactions cannot be nested in STM Haskell. For the sake of clarity, we consider that the set of function symbols ($\mathscr{F}$), lambda variables ($\mathscr{X}$), *TVar*s ($\mathscr{V}$) and data constructors ($\mathscr{K}$) underlying a program are finite sets that are pairwise distinct. Moreover, we suppose that $\mathscr{V} = \{t_1, ..., t_n\}$ is a totally ordered set. We also consider the existence of a mapping Δ from function names of $\mathscr{F}$ to expressions. In other words, Δ contains those function definitions that can be called from within a transaction.

A first kind of expressions, which we call *pure expressions* are those constituted by repeated application of only the rules $E1$ - $E7$. They correspond to the language defined in [47], being a classical functional language based on construction ($E2$), lambda abstraction ($E3$), application ($E5$), function calls ($E6$), and case expression ($E7$). Note the presence of *exceptions* ($E4$) – basically being the zero-arity predefined constructors `BAD` and `UNR`. A second kind of expressions, which we will call *STM expressions* are those that involve at least one application of a rule among $E8$ - $E11$. Retrieval of the content of a TVar ($E8$), updating the content of a TVar ($E9$), binding two STM expressions ($E10$) and defining the return expression ($E11$) are the main operations we can find in a STM expression.

A type system exists for STM expressions [6] and we will consequently suppose dealing only with well-typed expressions. For any expression e we will denote by e`::a` the fact that the expression is of type `a`. In particular, a STM expression having an outermost redex of the form $E8$, $E9$, $E10$ or $E11$ is of type `STM a` where `a` is the type of the expression returned when evaluating the STM expression, and a TVar as being of type `TVar a` where `a` is the type of the expression that we can store in this TVar. We will refer to STM expressions of type `STM a` as *STM operations*. We will also suppose implicitly that all considered expressions are closed, i.e. there is no free variable, and *well-formed*, that is to say that wherever `writeTVar` t e or `return` e appear in an expression, e is a pure expression.

In what follows, we suppose certain types and constructors given. Among them the type `Bool` defining the zero-arity constructors `True` and `False`, i.e. $\{$`True`,`False`$\} \in \mathscr{K}$. In the examples, we will furthermore use integers and lists, the latter being defined using two constructors: the zero-arity constructor `[]` to represent empty list and the binary constructor `(:)` which is often used in an infix way, i.e. `x:xs` is the list obtained by adding `x` in front of the list `xs`. Finally, to enhance readability of

f	$\in$	**Function Names** ($\mathscr{F}$)			
x, y	$\in$	**Lambda Variables** ($\mathscr{X}$)			
K	$\in$	**Data Constructors** ($\mathscr{K}$)			
t	$\in$	**Transactional Variables** ($\mathscr{V}$)			
e, p	$\in$	**Exp**	**Expressions**		
e, p	::=	x	variable		$[E1]$
	|	$K\ \overline{e}$	constructor	(value)	$[E2]$
	|	$\lambda x.e$	lambda abstraction	(value)	$[E3]$
	|	r	exception	(value)	$[E4]$
	|	$e_1\ e_2$	application		$[E5]$
	|	f	function call		$[E6]$
	|	`case` e `of` $\{alt_1 ... alt_n\}$	case-expression		$[E7]$
	|	`readTVar` t	STM read variable		$[E8]$
	|	`writeTVar` $t\ e$	STM write variable		$[E9]$
	|	e_1 `>>=` e_2	STM bind		$[E10]$
	|	`return` e	STM return	(value)	$[E11]$
r	$\in$	**Exceptions**			
r	::=	`BAD`			
	|	`UNR`			
alt	::=	$K\ \overline{x} \rightarrow e$			

Figure 5: Syntax of $\mathscr{HC}$ expressions

the examples, we will sometimes use the convenient so-called `do`-notation [15] and `let`-notation as a syntactic sugar:

$$\begin{aligned}
\texttt{let}\ x = e'\ \texttt{in}\ e &\equiv (\lambda x.e)\ e' \\
\texttt{do}\{x\texttt{<-}e;S\} &\equiv e\ \texttt{>>=}\ (\lambda x.\texttt{do}\{S\}) \\
\texttt{do}\{e;S\} &\equiv e\ \texttt{>>=}\ (\lambda _.\texttt{do}\{S\}) \\
\texttt{do}\{e\} &\equiv e
\end{aligned}$$

An expression is evaluated (or reduced) with respect to an *environment* σ, which is a mapping from *TVars* $t_i \in \mathscr{V}$ to pure expressions. We will denote by $\sigma(t_i)$ the (pure) expression that is associated to the *TVars* t_i in the environment σ. We will sometimes call this expression the *(transactional) value of* t_i. We will denote by $\sigma[t_i \mapsto e]$ the environment σ' such that $\sigma'(t_i) = e$ and $\forall t_k \in V/\{t_i\} : \sigma(t_k) = \sigma'(t_k)$. The rules by which an expression can be reduced are given in Fig. 6, in the form of a reduction relation [35] $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$ which, from the combination of an expression e and an environment σ returns a new expression e' and a new environment σ'. The reduction rules CALL and APP, where we denote the capture-avoiding substitution of e' for each free occurrence of x in e by $e[x/e']$, as well as CASE and BIND are standard rules for functional languages. Note that these reductions have no side-effect in the sense that σ is not modified. The rules READ and WRITE define the semantics of reading, respectively updating, a *TVar*. Note that each of these operations reduces to a return operation, allowing to bind their result with a second STM expression (BIND). The rule CTX allows a reduction to be processed in any *context* of the form $\mathbb{C}$ and the rule EXC allows to propagate an exception r, which, by the way, can be of any type following the context. As usual, we denote by $\rightarrow^*$ the reflexive-transitive closure of $\rightarrow$. To

$$\langle (\lambda x.e_1)\ e_2, \sigma\rangle \to \langle e_1[e_2/x], \sigma\rangle \quad (APP) \qquad \frac{f = e \in pgm}{\langle f, \sigma\rangle \to \langle e, \sigma\rangle} \quad (CALL)$$

$$\langle \texttt{case}\ K_i\ \overline{e_i}\ \texttt{of}\ \{..., K_i\ \overline{x_i} \to e, ...\}, \sigma\rangle \to \langle e[e_i/x_i], \sigma\rangle \quad (CASE)$$

$$\langle \texttt{readTVar}\ t, \sigma\rangle \to \langle \texttt{return}\ \sigma(t), \sigma\rangle \quad (READ)$$

$$\langle \texttt{writeTVar}\ t\ e, \sigma\rangle \to \langle \texttt{return}\ (), \sigma[t \mapsto e]\rangle \quad (WRITE)$$

$$\langle \texttt{return}\ e_1\ \texttt{>>=}\ e_2, \sigma\rangle \to \langle e_2\ e_1, \sigma\rangle \quad (BIND)$$

$$\frac{\langle e, \sigma\rangle \to \langle e', \sigma'\rangle}{\langle \mathbb{C}[\circ/e], \sigma\rangle \to \langle \mathbb{C}[\circ/e'], \sigma'\rangle} \quad (CTX) \qquad \langle \mathbb{C}[\circ/r], \sigma\rangle \to \langle r, \sigma\rangle \quad (EXC)$$

$$\mathbb{C} ::= \circ \mid \mathbb{C}\ e_2 \mid \texttt{case}\ \mathbb{C}\ \texttt{of}\ \{alt_1 ... alt_n\} \mid \mathbb{C}\ \texttt{>>=}\ e_2$$

Figure 6: Semantics of $\mathscr{H}$ expressions.

ease notation, when dealing with expressions other than STM operations, i.e. other than of type `STM a`, we will omit the environment, i.e. we write $e \to^* e'$ instead of $\langle e, \sigma\rangle \to^* \langle e', \sigma\rangle$, as the environment is never used nor modified when reducing a such expression. In this case, our semantics coincides with the semantics of the language defined in [47].

The attentive reader will notice that *TVar*s are considered as global variables: they can be accessed from everywhere in the program, but only through a direct reference. We will discuss the relevance of these and other limitations of $\mathscr{H}$ in Section 4.4.

In what follows, we suppose that expressions to be analyzed are processed beforehand in such a way that missing branches in a case expression are explicitly associated with a `BAD` exception. For example, if $\mathscr{K} = \{\texttt{True}, \texttt{False}\}$, the expression $\lambda x.\texttt{case}\ x\ \texttt{of}\ \{\texttt{True} \to f\}$ would be replaced by the expression $\lambda x.\texttt{case}\ x\ \texttt{of}\ \{\texttt{True} \to f, \texttt{False} \to \texttt{BAD}\}$. As in [47], we say that an expression crashes if it reduces to `BAD`.

Definition 1. *Let e be an expression and* σ *an environment, e* crashes *in* σ *iff* $\langle e, \sigma\rangle \to^* \langle \texttt{BAD}, \sigma'\rangle$.

In the particular case of pure expressions, we call an expression *crashfree* if and only if there is no way to make it crash (due to a missing pattern) [47]. More formally:

Definition 2. *Let e be a pure expression, e is* crashfree *iff* $\mathbb{C}[\circ/e] \not\to^* \texttt{BAD}$ *for any context* $\mathbb{C}$ *such that* `BAD` *does not appear syntactically in* $\mathbb{C}$.

In a similar vein, we say that an expression *diverges* if it cannot be reduced to a value, i.e. a lambda abstraction, a construction, a `return` or an exception, or if it reduces to `UNR`. The latter condition will turn out to be interesting in the context of the verification process.

Definition 3. *Let e be an expression and* σ *an environment, e* diverges *in* σ*, written* $\langle e, \sigma\rangle \uparrow^*$*, iff* $\langle e, \sigma\rangle \to^* \langle \texttt{UNR}, \sigma'\rangle$ *or there is no value val such that* $\langle e, \sigma\rangle \to^* \langle val, \sigma'\rangle$.

Again, in case of an expression of a type other than `STM a`, we will often omit the environment from the notation and simply write $e \uparrow^*$ to denote that e diverges.

4.2 Contracts

The syntax of contracts is given in Fig. 7. Contracts defined by application of only the rules $C1$ - $C4$ are reserved for specifying contracts on pure expressions and are identical to those defined in [47]. We call

them *pure contracts* in order to distinguish them from *STM contracts* which are contracts involving at least one application of the novel rule *C*5. Intuitively, we will associate pure contracts to pure expressions and STM contracts to STM expressions.

c	$\in$	**Contracts**		
c	$::=$	$\{x \mid p\}$	Predicate Contract	$[C1]$
	$\mid$	$x : c_1 \rightarrow c_2$	Dependent Function Contract	$[C2]$
	$\mid$	(c_1, c_2)	Data Constructor Contract	$[C3]$
	$\mid$	`Any`	Polymorphic Any Contract	$[C4]$
	$\mid$	$\parallel x : c_1 \Diamond c_2 \parallel c$	STM Operation Contract	$[C5]$

Figure 7: Syntax of contracts

We choose this syntax for the contract *C*5, which is called an *STM operation contract*, to fit with the type of STM operations, i.e. `STM a`. As we will see, the first part – $\parallel x : c_1 \Diamond c_2 \parallel$ – is related to the (software) transactional memory `STM` and the second – the contract c – to the expression returned, of type `a`. Like expressions, contracts are assumed to be well-typed. For the contract *C*5, we expect c to be the kind of contract which is typically associated to expressions of type `a`. Regarding c_1 and c_2, they should be contracts for expressions of type $(\mathtt{a_1}, ..., \mathtt{a_n})$ where $\mathtt{a}_i$ is the type of the expression stored in the *TVar*s t_i. This idea is expressed more formally by the typing rule in Fig. 8, which extends the typing system for contracts defined in [48]. Note that this implies that c, c_1 and c_2 are required to be pure contracts, as *TVar*s and returned expressions must be pure expressions.

$$\frac{\forall i : 1 \leq i \leq n : t_i :: \mathtt{TVar\ a_i}; c_1, c_2 :: (\mathtt{a_1}, ..., \mathtt{a_n}); c :: \mathtt{a}}{\parallel x : c_1 \Diamond c_2 \parallel c :: \mathtt{STM\ a}}$$

Figure 8: Typing rule for the STM operation contract.

The semantics of an expression *e satisfying* a contract c, denoted by $e \in c$ is defined in Fig. 9. The rules *CS*1 - *CS*4 are based on the original work from [47]. Intuitively, $e \in \{x \mid p\}$, where p is typically a boolean expression, if e is a sane expression (there is no proper way to make it crash) and the predicate $p[e/x]$ returns `True`. Note that the frequently used contract `Ok` is just a syntactic notation for a contract $\{x \mid \mathtt{True}\}$. An expression e satisfies $x : c_1 \rightarrow c_2$ if it satisfies c_2 when given an argument that satisfies c_1. Note the use of x which allows to refer from within e_2 to the value of the argument. Likewise, an expression e satisfies a pair of contracts if it evaluates to a pair and if each element satisfies its corresponding contract. As a pair is simply a particular constructor from $\mathscr{K}$ with a somewhat nonstandard notation, this contract can effectively be generalized to any constructor from $\mathscr{K}$. The special contract `Any` is satisfied by any pure expression, including crashing expressions such as `BAD`.

The definition *CS*5 is more particular, as it is dedicated to STM operations. Intuitively, a STM operation e satisfies $\parallel x : c_1 \Diamond c_2 \parallel c$ if, when it is performed with respect to an environment that satisfies c_1, it produces a new environment that satisfies c_2 and, moreover, it returns an expression satisfying c. In its definition, we use $\vec{\sigma}$ to refer to the pure expression $(e_1, ..., e_n)$ where $e_i = \sigma(t_i)$ for all $t_i \in \mathscr{V}$. Note also the use of x that allows to refer to the input environment both in c and c_2.

The attentive reader will notice that the satisfaction of a contract by an expression does not depend on specific requirements about the value of the *TVar*s, which is desirable as we target a static analysis. This is why we ensure that expressions will behave properly with respect to a given contract regardless the environment, which is incidentally omitted in *CS*1 - *CS*4. Also note that a diverging expression satisfies

$$
\begin{array}{lcll}
e \in \{x \mid p\} & \Leftrightarrow & e \text{ is pure and } (e \uparrow^* \text{ or } (e \text{ is crashfree and} & \\
 & & (p[e/x] \uparrow^* \text{ or } p[e/x] \rightarrow^* \texttt{True}))) & [CS1] \\
e \in x : c_1 \rightarrow c_2 & \Leftrightarrow & e \uparrow^* \text{ or } (e \rightarrow^* \lambda x.e_2 \text{ and} & \\
 & & \forall e_1 \in c_1 : (e\ e_1) \in c_2[e_1/x]) & [CS2] \\
e \in (c_1, c_2) & \Leftrightarrow & e \uparrow^* \text{ or } (e \rightarrow^* (e_1, e_2) \text{ and} & \\
 & & e_1 \in c_1, e_2 \in c_2) & [CS3] \\
e \in \texttt{Any} & \Leftrightarrow & e \text{ is pure.} & [CS4] \\
e \in \parallel x : c_1 \Diamond c_2 \parallel c & \Leftrightarrow & \forall \sigma : \vec{\sigma} \in c_1 : \langle e, \sigma \rangle \uparrow^* & \\
 & & \text{or } (\langle e, \sigma \rangle \rightarrow^* \langle \texttt{return}\ e', \sigma' \rangle \text{ and} & \\
 & & \vec{\sigma'} \in c_2[\vec{\sigma}/x] \text{ and } e' \in c[\vec{\sigma}/x]) & [CS5]
\end{array}
$$

Figure 9: Contract Satisfaction

any contract and that the expression p in a contract of the form $\{x \mid p\}$ can also diverge. This means that our framework only concerns *partial correctness* and should be paired with a termination analysis [24, 14] in order to obtain results concerning total correction.

In Fig. 10, we show an example of a function with a contract such that it is satisfied by the function definition. Note that, as for the others examples, we use here a convenient Haskell-like syntax, which is also the one we use in our prototype, instead of our more formal but equivalent syntax. Intuitively, the point of this function is to store a message into a box – which is represented by the TVar `box`, to log this action by incrementing a counter – another TVar `ct`, and to return the message that was previously stored in the box. This operation is allowed only if one are connected – i.e. if the expression stored in the TVar `c` reduces to `True`. As such, the first part of the contract depicts the fact that there is no specific requirement about the message, apart from being crashfree. In the second part, we require the TVar `c` to contain `True`, and we say that, if this requirement holds, the transaction modifies the *TVar*s in a such way that `c` is still `True` (we are still connected), the counter `ct` has been increased, and the expression returned correspond the initial expression stored in `box`. Note that, while this should not disturb the reader, we have used some syntactic sugar in order to allow tuples of variable in the left part of predicate contracts, i.e. $\{(x_1,...,x_n) \mid p\}$ as a shorthand for $\{x \mid \texttt{case}\ x\ \texttt{of}\ (x_1,...x_n) \rightarrow p\}$, and to avoid repeating variable(s) in (dependent) contracts when it is not useful, i.e. $\parallel \{x \mid p\} \Diamond c_2 \parallel c$ as a shorthand for $\parallel x : \{x \mid p\} \Diamond c_2 \parallel c$.

```
send :: Msg -> STM Msg
send :: Ok  -> || {(c, box, ct ) | c} <> {(c',box',ct') | c' && ct'>ct} || {res|res==box}
send msg = do connected <- readTVar c
              case connected of
                True  -> do oldMsg <- readTVar box
                            writeTVar box msg
                            x <- readTVar ct
                            writeTVar ct (x+1)
                            return oldMsg
                False -> BAD
```

Figure 10: The function `send` and its type and contract.

Note that our extensions to the contract system of [47] are such that the set of desirable properties,

discussed in the latter work, still hold. For instance, we can still deal with function calls and recursion in a modular way: to check that the definition of a function f satisfies the contract c_f where f is defined as $f = e$ and e is an expression containing a call to function g, we can simply check that $\lambda g.e \in c_g \to c_f$ where c_g is the contract of g.

Now that we have introduced the basic formalism allowing to specify contracts over expressions, we can now formally define what it means for a contract to be a *transactional invariant*:

Definition 4. *An STM operation e is* consistent *with respect to a contract c iff $e \in \| \; c \diamondsuit c \; \|$ `Any`. The contract c is then called an* transactional invariant *of e.*

For example, the contract $\{t \mid t > 0\}$ is a transactional invariant of the STM operation

$$e = \texttt{readTVar}\ t\ \texttt{>>=}\ \lambda x.\texttt{writeTVar}\ t\ (x+1)$$

To conclude this section, let us briefly restate the basic idea behind the verification process: first, the programmer writes a contract c that corresponds to his or her view of the consistency over the *TVar*s, and, possibly, contracts for the functions defined by either pure or STM expressions. Secondly, it needs to be verified whether the contract c is effectively a transactional invariant for every transactions appearing in the program, a transaction being the STM operation e where `atomically(`e`)` appears in the program. Every transaction for which c cannot be proven to be a transactional invariant possibly represents an application-level race condition. Before formalizing the verification step, note that our definitions imply that a transaction be a closed expression, i.e. it may not contain free variables. If there *are* free variables in a transaction e, rather than verifying $e \in \| \; c \diamondsuit c \; \|$ `Any`, one must verify whether $(\lambda x_1 ... \lambda x_n.e) \in c_1 \to \cdots \to c_n \to \| \; c \diamondsuit c \; \|$ `Any`, where $x_1, ..., x_n$ are the free variables in e and $c_1, ..., c_n$ the contracts associated to $x_1, ..., x_n$. This is what was already illustrated by the example at the end of Section 3, where the contract `Ok` was associated to the (initially) free variable `n`. Note that, in practice, these contracts can be provided by the programmer indirectly, through a top-level contract attribution, or – to some extent – be automatically generated with by-default value [47]. This, however, is out of the scope of the current work.

4.3 Checking through Program and Contract Transformation

Our approach in checking whether a program written in STM Haskell is free of application-level race conditions consists then in transforming a transaction e and its contract c into a pure expression e' and a pure contract c', in a such way that $e' \in c'$ implies $e \in c$. This transformation, represented by the $\mathscr{T}$-operator, is defined – for expressions – in Fig. 11. For sake of clarity, the transformation assumes that only a single TVar is handled by the program, i.e. $\mathscr{V} = \{t\}$, but the definitions can readily be extended towards handling a given set of *TVar*s.

The basic idea is to transform a STM operation e returning e' and possibly updating the (transactional) value of t into a lambda expression $\lambda t.(e', e'')$ where e'' is the updated value of t. Reading the TVar is modeled by a function that associates to the initial value of t the couple (t, t). This depicts the fact that the operation returns the value stored in the TVar (the first t) while the value stored in t does not change (the second t). The update operation does not return a relevant expression (so we return the nullary constructor `()`, as usual in Haskell) but it replaces the expression that was previously stored in t by the given expression. Note that, as we consider well-formed expressions, the expression e that will be written into the TVar is pure, and hence need not be transformed. When a STM operation is binded with another STM expression, both expressions are converted and the return expression and the updated TVar value of the first transformed expression are provided as input for application with the second transformed

$$
\begin{array}{lcll}
\mathscr{T}(\texttt{readTVar}\ t) & = & \lambda t.(t,t) & \\
\mathscr{T}(\texttt{writeTVar}\ t\ e) & = & \lambda t.(\texttt{()},e) & \\
\mathscr{T}(e_1\ \texttt{>>=}\ e_2) & = & \lambda t.(\mathscr{T}(e_2)\ (\texttt{fst}\ e_1')\ (\texttt{snd}\ e_1')) & \text{where} \quad e_1' = \mathscr{T}(e_1)\ t \\
 & & & \qquad\qquad\ \ \texttt{fst} = \lambda(a,b).a \\
 & & & \qquad\qquad\ \ \texttt{snd} = \lambda(a,b).b \\
\mathscr{T}(\texttt{return}\ e) & = & \lambda t.(e,t) & \\
\mathscr{T}(f) & = & \lambda t.(f\ t) & \text{if } f :: \texttt{STM a}. \\
 & = & f & \text{else}. \\
\mathscr{T}(e_1\ e_2) & = & \lambda t.(\mathscr{T}(e_1)\ \mathscr{T}(e_2)\ t) & \text{if } e_1\ e_2 :: \texttt{STM a}. \\
 & = & \mathscr{T}(e_1)\ \mathscr{T}(e_2) & \text{else}. \\
\mathscr{T}(\lambda x.e) & = & \lambda x.\mathscr{T}(e) & \\
\mathscr{T}(K\ \overline{e}) & = & K\ \overline{\mathscr{T}(e)} & \\
\mathscr{T}(x) & = & x & \\
\mathscr{T}(r) & = & r & \\
\mathscr{T}(e_{\texttt{case}}) & = & \lambda t.((\texttt{case}\ \mathscr{T}(e)\ \texttt{of}\ \overline{pat_i \to \mathscr{T}(e_i)})\ t) & \text{if } e_{\texttt{case}} :: \texttt{STM a}. \\
 & = & \texttt{case}\ \mathscr{T}(e)\ \texttt{of}\ \overline{pat_i \to \mathscr{T}(e_i)} & \text{else}.
\end{array}
$$

where $e_{\texttt{case}} \equiv \texttt{case}\ e\ \texttt{of}\ \overline{pat_i \to e_i}$

Figure 11: $\mathscr{T}$-transformation for $\mathscr{H}$ expressions

expression. Note the use of `fst` and `snd` that retract, respectively, the first and the second expressions from a pair. To help understanding the intuition behind this transformation, we can see that

$$\mathscr{T}(\texttt{readTVar}\ t\ \texttt{>>=}\ \lambda x.\texttt{writeTVar}\ t\ (x+1))$$
$$=$$
$$\lambda t.((\lambda x.\lambda t.(\texttt{()},x+1))\ (\texttt{fst}\ ((\lambda t.(t,t))\ t))\ (\texttt{snd}\ ((\lambda t.(t,t))\ t))$$

which can *symbolically* [22] be rewritten into $\lambda t.(\texttt{()},t+1)$, clearly reflecting the update of t by $t+1$. Transforming a return operation is more straightforward as it does not change the value of the *TVar*. Transforming the remaining expressions basically boils down to propagating the transformation to their subexpressions, as the latter may contain STM operations. Note that our transformation $\mathscr{T}$-operator is defined such that:

- if e is a STM operation, then the execution of $\mathscr{T}(e)\ e'$ gives a couple (e_1,e_2) where e_1 is the expression that would be returned by the STM operation e, and e_2 is the expression that would be stored finally in the TVar if the latter would have contained e' before performing the STM operation.
- if e is a pure expression, then the execution of $\mathscr{T}(e)$ will produce the same result as the execution of e.

Moreover, the $\mathscr{T}$-operator can be generalized easily for a (totally ordered) set of n *TVar*s, i.e. $\mathscr{V} = \{t_1,...,t_n\}$. This would require transforming into an expression that takes as argument a tuple $(t_1,...,t_n)$ instead of a single t in Fig 11 and that returns a couple where the second element is a new tuple of n expressions. For example, the first rule would look like:[2]

$$\mathscr{T}(\texttt{readTVar}\ t_k) \quad = \quad \lambda(t_1,...,t_k,...,t_n).(t_k,(t_1,...,t_k,...,t_n))$$

[2] we use the following syntactic sugar : $\lambda(t_1,...t_n).e \equiv \lambda x.\texttt{case}\ x\ \texttt{of}\ (t_1,...,t_n) \to e$ where x is not a free variable of e.

We also need a technique to convert STM contracts into pure contracts. For this purpose, we override the $\mathscr{T}$-operator such that it deals with contracts too. The transformation is rather straightforward and depicted in Fig. 12. The main point is the conversion of a STM operation contract into a dependant function contract in order to fit with the form of a transformed STM operation.

$$\begin{array}{rcl}
\mathscr{T}(\{x \mid p\}) & = & \{x \mid p\} \\
\mathscr{T}(x : c_1 \to c_2) & = & x : \mathscr{T}(c_1) \to \mathscr{T}(c_2) \\
\mathscr{T}(c_1, c_2) & = & (\mathscr{T}(c_1), \mathscr{T}(c_2)) \\
\mathscr{T}(\texttt{Any}) & = & \texttt{Any} \\
\mathscr{T}(\parallel x : c_1 \Diamond c_2 \parallel c) & = & x : c_1 \to (c, c_2)
\end{array}$$

Figure 12: $\mathscr{T}$-transformation for contracts

Fig. 13 enlists a number of easily proven but important properties of the $\mathscr{T}$-operator. Property (1) states that the resulting expression, or contract, is pure in the sense that no STM-related construction remains after the transformation. This follows immediately from the definition of $\mathscr{T}$ and the fact that we consider only well-formed contracts, i.e. subcontracts of STM contracts are pure. Secondly, $\mathscr{T}$ is an idempotent operator (2) and it does not change its argument when the latter is already pure (3 - 4). Further, it follows that the transformation of a STM operation different from `BAD` and `UNR` always reduces to a lambda abstraction (5). The two last properties (6) and (8) which can be proved by induction on e, are fundamental in our framework and depict the equivalence that links the semantics of a STM operation with its transformed pure counterpart.

- $\mathscr{T}(e)$, resp. $\mathscr{T}(c)$, is a pure expression, resp. contract. (1)
- $\mathscr{T}(\mathscr{T}(e)) \equiv \mathscr{T}(e), \mathscr{T}(\mathscr{T}(c)) \equiv \mathscr{T}(c)$ (2)
- $\mathscr{T}(e) \equiv e$ if e is a pure expression. (3)
- $\mathscr{T}(c) \equiv c$ if c is a pure contract. (4)
- $\mathscr{T}(e) \to^* \lambda x.e'$ if $e :: \texttt{STM a}$ and $e \not\equiv r$. (5)
- $\langle e, \sigma \rangle \uparrow^* \iff \mathscr{T}(e)\ \vec{\sigma} \uparrow^*$ for $e :: \texttt{STM a}$ (6)
- $\langle e, \sigma_1 \rangle \to^* \langle \texttt{return}\ e', \sigma_2 \rangle \iff \mathscr{T}(e)\ \vec{\sigma_1} \to^* (e', \vec{\sigma_2})$ (7)
  for $e :: \texttt{STM a}$ (8)

Figure 13: Properties of $\mathscr{T}$

Trivially, as a direct consequence of properties (3) and (4), we have that $e \in c \iff \mathscr{T}(e) \in \mathscr{T}(c)$ when e and c are a pure expression and contract. More importantly, the same property holds for a STM operation and a STM operation contract of same type, as stated by the following theorem.

Theorem 1. *Let e be an expression of type* `STM a` *and let c be a contract of type* `STM a`*,*

$$e \in c \iff \mathscr{T}(e) \in \mathscr{T}(c)$$

Proof. As c is of type `STM a`, it is a contract of the form $c \equiv \parallel c_1 \Diamond c_2 \parallel c'$. We distinguish three cases:

- $e \equiv \texttt{BAD}$: We can see, by the semantics of contracts and expressions, that $\texttt{BAD} \notin \parallel c_1 \Diamond c_2 \parallel c'$ and $\mathscr{T}(\texttt{BAD}) \equiv \texttt{BAD} \notin x : c_1 \to (c', c_2) \equiv \mathscr{T}(\parallel c_1 \Diamond c_2 \parallel c')$ for any c_1, c_2, c'.
- $e \equiv \texttt{UNR}$: Similarly, $\texttt{UNR} \in \parallel c_1 \Diamond c_2 \parallel c'$ and $\mathscr{T}(\texttt{UNR}) \equiv \texttt{UNR} \in x : c_1 \to (c', c_2) \equiv \mathscr{T}(\parallel c_1 \Diamond c_2 \parallel c')$ for any c_1, c_2, c'.

- $e \not\equiv \texttt{BAD}, \texttt{UNR}$:

$$
\begin{array}{ll}
 & \mathscr{T}(e) \in \mathscr{T}(c) \\
\iff & \text{(form of } c\text{)} \\
 & \mathscr{T}(e) \in \mathscr{T}(\parallel c_1 \Diamond c_2 \parallel c') \\
\iff & \text{(def. of } \mathscr{T}\text{)} \\
 & \mathscr{T}(e) \in x : c_1 \to (c', c_2) \\
\iff & \text{(def. of } \in \text{ + prop. (5))} \\
 & \forall e_1 \in c_1 : (\mathscr{T}(e)\ e_1) \in (c', c_2)[e_1/x] \\
\iff & \text{(def. of } \sigma \text{ + type of } c_1\text{)} \\
 & \forall \sigma : \vec{\sigma} \in c_1 : (\mathscr{T}(e)\ \vec{\sigma}) \in (c', c_2)[\vec{\sigma}/x] \\
\iff & \text{(def. of } \in \text{ + type of } c_2\text{)} \\
 & \forall \sigma : \vec{\sigma} \in c_1 : \mathscr{T}(e)\ \vec{\sigma} \uparrow^* \text{ or } ((\mathscr{T}(e)\ \vec{\sigma}) \to^* (e', \overrightarrow{\sigma'}) \\
 & \text{ and } e' \in c'[\vec{\sigma}/x], \overrightarrow{\sigma'} \in c_2[\vec{\sigma}/x]) \\
\iff & \text{(prop. (6) and (8))} \\
 & \forall \sigma : \vec{\sigma} \in c_1 : \langle e, \sigma \rangle \uparrow^* \text{ or } (\langle e, \sigma \rangle \to^* \langle \texttt{return}\ e', \sigma' \rangle \\
 & \text{ and } e' \in c'[\vec{\sigma}/x]), \overrightarrow{\sigma'} \in c_2[\vec{\sigma}/x] \\
\iff & \text{(def. of } \mathscr{T}\text{)} \\
 & e \in \parallel c_1 \Diamond c_2 \parallel c' \\
\iff & \text{(form of } c\text{)} \\
 & e \in c
\end{array}
$$

□

The following corollary generalizes the above result towards any expression and contract (of the same type). It can be easily proven by induction on the structure of the expression and corresponding contract, using the results for a pure subexpression (contract) and a subexpression (contract) of type `STM a` as base cases.

Corollary 1. *Let e be an expression of type* `a` *and let* `c` *be a contract of type a,*

$$e \in c \iff \mathscr{T}(e) \in \mathscr{T}(c)$$

The above corollary basically states correctness of approach: no precision is lost by transforming a STM Haskell transaction into a pure function and using standard techniques for contract checking [47] to prove its consistence with respect to the transactional invariant. The approach has been fully implemented and the concerned reader is invited to try the prototype of our framework[3].

4.4 Limitations and extensions

Blocking and Composable Transactions

STM Haskell also allows to define blocking and composable (alternatives) STM operations by means of the primitives `retry :: STM a` and `orElse :: STM a -> STM a -> STM a` [15]. The first one

[3]Our prototype can be downloaded at the following URL:
`http://www.info.fundp.ac.be/~rde`

is an outstandingly simple way to force a thread to wait for an event. Semantically, `retry` just make the transaction abort, i.e. all changes are discarded, and the transaction is restarted from the beginning (in practice, it restarts only when a related TVar is modified, but this is only an implementation detail). For example, in the following call, the transaction will conceptually wait for the TVar `connected` to be `True` before continuing to the operation `sendMessage`.

```
atomically ( do c <- readTVar connected
                case c of True  -> sendMessage
                          False -> retry )
```

There is no difficulty to deal with `retry` in our framework. It is sufficient to define $\mathscr{T}(\texttt{retry}) = \texttt{UNR}$. Indeed, recall that we do not have to deal with diverging expressions, nor with the values of the *TVar*s at the intermediate states of the transaction, but only with the possible values of the *TVar*s at the very end of the transaction, which are not influenced by a `retry` branch.

The other primitive, `orElse`, allows multiple STM operations to be composed as alternatives. Basically, e_1 `` `orElse` `` e_2 proceeds as follows: first, e_1 is evaluated. If its evaluation does *not* result in `retry`, the operation ends with the result computed by e_1. If on the other hand, e_1 *does* evaluate to `retry`, rather than restarting the operation, e_2 is evaluated. Only if the latter also result in `retry` is the entire operation restarted.

In order to verify a transaction e containing an `orElse` operation, it suffices to verify multiple versions of the transaction, say $\Gamma(e)$, one version for each possible combination of alternative evaluations. The Γ-operator computing all possible such evaluations is partly defined below:

$$\begin{aligned}
\Gamma(e_1 \text{ `orElse` } e_2) &= \Gamma(e_1) \cup \Gamma(e_2) \\
\Gamma(e_1 \texttt{ >>= } e_2) &= \{e_1' \texttt{ >>= } e_2' \mid e_1' \in \Gamma(e_1), e_2' \in \Gamma(e_2)\} \\
\Gamma(\lambda x.e) &= \{\lambda x.e' \mid e' \in \Gamma(e)\} \\
&\ldots
\end{aligned}$$

Instead of verifying $e \in c$, we have then to verify that $\forall e' \in \Gamma(e) : e' \in c$.

*TVar*s as arguments of a function

One limitation of our language $\mathscr{H}$ is that, contrary to full STM Haskell, it does *not* allow a function (or, more precisely, a lambda abstraction) to have *TVar*s as its arguments. However, as long as we consider a fixed set of *TVar*s manipulated by the program, this limitation can be overcome by using well-known techniques from program specialization [21, 19] in order to specialize both the function and its corresponding contract with respect to all possible subsets of *TVar*s that can be provided as actual arguments in a call to the function. Let us discuss this idea informally by looking at a simple example. Consider the following function, which takes an argument `x` supposed to be a TVar containing an integer, and which updates the given TVar by incrementing this value.

```
f :: TVar Int -> STM ()
f x = do n <- readTVar x
         writeTVar x (n+1)
```

Suppose that we want to express in a contract that this function must be called with respect to a TVar containing a positive value, and that the resulting value after the update must be strictly greater than the initial value. This could be expressed by a new kind of contract such as the following in which `t` and `t'` are used to refer to the initial, respectively, final value of the function's argument:

```
f :: TVar[t,t'] -> | t >= 0 <> t' > t | Any
```

Supposing now that the program manipulates two *TVar*s, say $\mathcal{V} = \{\texttt{tA}, \texttt{tB}\}$, we can generate *two* versions of the function and its associated contract: one explicitly referring to `tA`, the other to `tB`. Let us take, for example, the result of specializing w.r.t. to `tA` (the result for `tB` is of course similar):

```
f_tA :: || { (tA,tB) | tA >= 0 } <> { (tA',tB') | tA' >= tA } || Any
f_tA = do n <- readTVar tA
          writeTVar tA (n+1)
```

Then, every call to `f tA` will be replaced by a call to `f_tA`. The specialized function no longer contains a TVar as argument, and hence can be verified (w.r.t. to the specialized contract) by our standard framework.

Note that this solution works as long as the set of manipulated *TVar*s is fixed (there are no dynamically created *TVar*s), and are not aggregated into a (possibly recursive) data structure. Whether the technique can be adapted to these situations is an interesting topic for further research.

5 Conclusion and Related Work

Data races being one of the most common sources of concurrency bugs, a lot of static [3, 33, 44, 11, 40, 17, 41, 39, 12, 7, 20] and dynamic [31, 42, 36, 49, 10, 37] analysis techniques have been developed to detect them [34] over the last couple of decades. The other categories of concurrency bugs discussed broadly in the literature are atomicity violation [13, 27, 30] and locking-related bugs [42]. However, very few attention has been given to more higher-level kind of bugs, such as those involving multiple variables, while studies have pointed out that they represent one third of the non-deadlock concurrency bugs found in real-world programs [26]. Moreover, we are convinced that, when using higher-level synchronization techniques/concurrent languages – like STM Haskell – which allow to avoid problems related to atomicity or locks, the proportion of those bugs involving multiple variables is greatly increased.

Several frameworks try to tackle the problem of multiple variables [25, 29, 28, 1, 45, 38]. In those works, static or dynamic techniques are used in order to infer possible correlations between shared variables. Then, they allow to detect races related to a set of correlated variables. This idea is depicted by the concept of *high-level data race*, introduced by [1]. Such a race occurs when a set of shared memory locations is meant to be accessed atomically, but the memory locations are accessed separately somewhere in the program [45, 38]. The interest of these tools is indisputable as they find a lot of races without requiring (a lot of) annotations, but they typically can trigger false positives and false negatives. Indeed, two shared variables can be accessed together but in a bad way while they can also be accessed separately without necessarily violating the link between them. In other words, while finding the adequate granularity of the atomic region is necessary, the *effect* of this atomic region on the data also matters.

To overcome this limitation, we have to deal with the subtly variant concept of *application-level race condition*, which is introduced and handled (to the best of our knowledge) exclusively by [4]. This notion not only captures the existence of a link between shared variables, but *explicitly* involves the *nature* of that link. By defining under what condition another thread can violate a program invariant, detection of application-level race conditions is more accurate, avoiding a lot of false positives and negatives. On the other hand, analyzes are typically heavier to use as the programmer has to provide additional annotations. Although the goal of our work is similar to the static analysis developed for verifying atomic blocks in an object-oriented language [4], the method to achieve that goal is very different. This is not not only because we need to address distinct language-related issues such as the particular control-flow inherent

to higher-order functional programming in our case, or access permissions and unpacking methodologies related to object-oriented programming in [4]. A more fundamental difference is that, in our framework, the consistency definition is written using *the same language* as the program, while [4] relies on a distinct formalism, *typestate* [8]. As a result, we feel our approach is both more convenient and expressive as we use full Haskell to express our invariants.

As we have basically shown how the problem of detecting application-level race conditions can be recasted in the setting of contract checking for non-concurrent programs [47, 5], it is to be expected that recent improvements for rendering the verification of such contracts more practical [46] will have a positive influence on the number and kind of races that can be detected by our approach.

Finally, let us note that consistency of STM in Haskell can also to some extent be checked dynamically, by using the language primitive `always` (previously called `check`) [16]. For our running example, we could write the following property that would be checked at the end of every transactions performed:

```
always ( do tab <- readTVar shTab
            s <- readTVar shSum
            return (sum tab == s) )
```

While guaranteeing consistency during the program's execution, being a dynamic technique, it cannot be used to statically prove that the program is application-level race condition free. It would be interesting to see to what extent such dynamic verification can be coupled with static checking in order to improve the detection of such race conditions and/or perform a more detailed error reporting.

Acknowledgments. We thank the anonymous reviewers for their constructive comments on a previous version of this paper.

References

[1] Cyrille Artho, Klaus Havelund & Armin Biere (2003): *High-Level Data Races*. In: *NDDL/VVEIS*, pp. 82–93, doi:10.1002/stvr.281.

[2] Mike Barnett, K. Rustan M. Leino & Wolfram Schulte (2005): *The spec# programming system: an overview*. In: *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, Springer-Verlag, Berlin, Heidelberg, pp. 49–69, doi:10.1007/978-3-540-30569-9-3.

[3] Nels E. Beckman (2006): *A Survey of Methods for Preventing Race Conditions*.

[4] Nels E. Beckman, Kevin Bierhoff & Jonathan Aldrich (2008): *Verifying correct usage of atomic blocks and typestate*. In Gail E. Harris, editor: *OOPSLA*, ACM, pp. 227–244, doi:10.1145/1449764.1449783.

[5] Matthias Blume & David McAllester (2006): *Sound and complete models of contracts*. *J. Funct. Program.* 16(4-5), pp. 375–414, doi:10.1017/S0956796806005971.

[6] Johannes Borgstrom, Karthikeyan Bhargavan & Andrew D. Gordon (2009): *A compositional theory for STM Haskell*. In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, ACM, New York, NY, USA, pp. 69–80, doi:10.1145/1596638.1596648.

[7] Chandrasekhar Boyapati, Robert Lee & Martin Rinard (2002): *Ownership types for safe programming: preventing data races and deadlocks*. *SIGPLAN Not.* 37(11), pp. 211–230, doi:10.1145/583854.582440.

[8] Robert DeLine & Manuel Fähndrich (2004): *Typestates for Objects*. In Martin Odersky, editor: *ECOOP 2004 – Object-Oriented Programming*, *Lecture Notes in Computer Science* 3086, Springer Berlin Heidelberg, pp. 465–490, doi:10.1007/978-3-540-24851-4-21.

[9] Romain Demeyer & Wim Vanhoof (2012): *A Framework for Verifying the Application-Level Race-Freeness of Concurrent Programs*. Available at `http://users.dsic.upv.es/workshops/wlpe2012/accepted_papers.html`.

[10] Tayfun Elmas, Shaz Qadeer & Serdar Tasiran (2007): *Goldilocks: a race and transaction-aware java runtime*. *SIGPLAN Not.* 42(6), pp. 245–255, doi:10.1145/1273442.1250762.

[11] Dawson Engler & Ken Ashcraft (2003): *RacerX: effective, static detection of race conditions and deadlocks*. *SIGOPS Oper. Syst. Rev.* 37, pp. 237–252, doi:10.1145/1165389.945468.

[12] Cormac Flanagan & Stephen N. Freund (2000): *Type-based race detection for Java*. *SIGPLAN Not.* 35, pp. 219–232, doi:10.1145/358438.349328.

[13] Cormac Flanagan & Shaz Qadeer (2003): *A type and effect system for atomicity*. *SIGPLAN Not.* 38, pp. 338–349, doi:10.1145/780822.781169.

[14] Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp & René Thiemann (2006): *Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages*. In Frank Pfenning, editor: *Term Rewriting and Applications*, *Lecture Notes in Computer Science* 4098, Springer Berlin Heidelberg, pp. 297–312, doi:10.1007/11805618_23.

[15] Tim Harris, Simon Marlow, Simon Peyton-Jones & Maurice Herlihy (2005): *Composable memory transactions*. In: *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, New York, NY, USA, pp. 48–60, doi:10.1145/1065944.1065952.

[16] Tim Harris & Simon Peyton Jones (2006): *Transactional memory with data invariants*.

[17] Thomas A. Henzinger, Ranjit Jhala & Rupak Majumdar (2004): *Race checking by context inference*. *SIGPLAN Not.* 39, pp. 1–13, doi:10.1145/996893.996844.

[18] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain & John Peterson (1992): *Report on the programming language Haskell: a non-strict, purely functional language version 1.2*. *SIGPLAN Not.* 27(5), pp. 1–164, doi:10.1145/130697.130699.

[19] John Hughes (1996): *An Introduction to Program Specialisation by Type Inference*. Glasgow University. Published electronically.

[20] Bart Jacobs, Frank Piessens, Jan Smans, K. Rustan M. Leino & Wolfram Schulte (2008): *A programming model for concurrent object-oriented programs*. *ACM Trans. Program. Lang. Syst.* 31(1), pp. 1:1–1:48, doi:10.1145/1452044.1452045.

[21] Neil D. Jones, Carsten K. Gomard & Peter Sestoft (1993): *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[22] Simon L. Peyton Jones (1996): *Compiling Haskell by program transformation: a report from the trenches*.

[23] Simon Peyton Jones (2007): *Beautiful Concurrency*. Ch. 24.

[24] Ruslán Ledesma-Garza & Andrey Rybalchenko (2012): *Binary Reachability Analysis of Higher Order Functional Programs*. In Antoine Miné & David Schmidt, editors: *Static Analysis*, *Lecture Notes in Computer Science* 7460, Springer Berlin Heidelberg, pp. 388–404, doi:10.1007/978-3-642-33125-1-26.

[25] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa & Yuanyuan Zhou (2007): *MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs*. *SIGOPS Oper. Syst. Rev.* 41, pp. 103–116, doi:10.1145/1323293.1294272.

[26] Shan Lu, Soyeon Park, Eunsoo Seo & Yuanyuan Zhou (2008): *Learning from mistakes: a comprehensive study on real world concurrency bug characteristics*. In Susan J. Eggers & James R. Larus, editors: *ASPLOS*, ACM, pp. 329–339, doi:10.1145/1346281.1346323.

[27] Shan Lu, Joseph Tucek, Feng Qin & Yuanyuan Zhou (2006): *AVIO: detecting atomicity violations via access interleaving invariants*. *SIGPLAN Not.* 41(11), pp. 37–48, doi:10.1145/1168918.1168864.

[28] Brandon Lucia & Luis Ceze (2009): *Finding concurrency bugs with context-aware communication graphs*. In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, ACM, New York, NY, USA, pp. 553–563, doi:10.1145/1669112.1669181.

[29] Brandon Lucia, Luis Ceze & Karin Strauss (2010): *ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations*. *SIGARCH Comput. Archit. News* 38(3), pp. 222–233, doi:10.1145/1816038.1815988.

[30] Brandon Lucia, Joseph Devietti, Karin Strauss & Luis Ceze (2008): *Atom-Aid: Detecting and Surviving Atomicity Violations*. *SIGARCH Comput. Archit. News* 36(3), pp. 277–288, doi:10.1145/1394608.1382145.

[31] Daniel Marino, Madanlal Musuvathi & Satish Narayanasamy (2009): *LiteRace: effective sampling for lightweight data-race detection*. In: *In PLDI*, doi:10.1145/1542476.1542491.

[32] Bertrand Meyer (1992): *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[33] Mayur Naik, Alex Aiken & John Whaley (2006): *Effective static race detection for Java*. *SIGPLAN Not.* 41(6), pp. 308–319, doi:10.1145/1133255.1134018.

[34] Robert H. B. Netzer (1993): *Optimal tracing and replay for debugging shared-memory parallel programs*. *SIGPLAN Not.* 28(12), pp. 1–11, doi:10.1145/174267.174268.

[35] Flemming Nielson, Hanne Riis Nielson & Chris Hanking (2005): *Principles of Program Analysis*. Springer.

[36] Robert O'Callahan & Jong-Deok Choi (2003): *Hybrid dynamic data race detection*. *SIGPLAN Not.* 38(10), pp. 167–178, doi:10.1145/966049.781528.

[37] Chang-Seo Park & Koushik Sen (2008): *Randomized active atomicity violation detection in concurrent programs*. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, ACM, New York, NY, USA, pp. 135–145, doi:10.1145/1453101.1453121.

[38] Vasco Pessanha, Ricardo J. Dias, Jo ao M. Louren¸co, Eitan Farchi & Diogo Sousa (2011): *Practical verification of high-level dataraces in transactional memory programs*. In: *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '11, ACM, New York, NY, USA, pp. 26–34, doi:10.1145/2002962.2002968.

[39] Polyvios Pratikakis, Jeffrey S. Foster & Michael Hicks (2006): *LOCKSMITH: context-sensitive correlation analysis for race detection*. *SIGPLAN Not.* 41(6), pp. 320–331, doi:10.1145/1133255.1134019.

[40] Shaz Qadeer & Dinghao Wu (2004): *KISS: keep it simple and sequential*. *SIGPLAN Not.* 39(6), pp. 14–24, doi:10.1145/996893.996845.

[41] Amit Sasturkar, Rahul Agarwal, Liqiang Wang & Scott D. Stoller (2005): *Automated type-based analysis of data races and atomicity*. In: *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, ACM, New York, NY, USA, pp. 83–94, doi:10.1145/1065944.1065956.

[42] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro & Thomas Anderson (1997): *Eraser: a dynamic data race detector for multithreaded programs*. *ACM Trans. Comput. Syst.* 15(4), pp. 391–411, doi:10.1145/265924.265927.

[43] Nir Shavit & Dan Touitou (1995): *Software transactional memory*, pp. 204–213. doi:10.1145/224964.224987.

[44] Nicholas Sterling (1993): *WARLOCK - A Static Data Race Analysis Tool*.

[45] Bruno Teixeira, Jo ao Louren¸co, Eitan Farchi, Ricardo Dias & Diogo Sousa (2010): *Detection of Transactional Memory anomalies using static analysis*. In: *Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '10, ACM, New York, NY, USA, pp. 26–36, doi:10.1145/1866210.1866213.

[46] Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen & Dan Rosén (2013): *HALO: haskell to logic through denotational semantics*. In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, ACM, New York, NY, USA, pp. 431–442, doi:10.1145/2429069.2429121.

[47] Dana N. Xu, Simon Peyton Jones & Koen Claessen (2009): *Static contract checking for Haskell*. *SIGPLAN Not.* 44(1), pp. 41–52, doi:10.1145/1594834.1480889.

[48] Na Xu (2008): *Static contract checking for Haskell*. Technical Report UCAM-CL-TR-737, University of Cambridge, Computer Laboratory. Available at `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-737.pdf`.

[49] Yuan Yu, Tom Rodeheffer & Wei Chen (2005): *RaceTrack: efficient detection of data race conditions via adaptive tracking*. *SIGOPS Oper. Syst. Rev.* 39, pp. 221–234, doi:10.1145/1095809.1095832.