Static Application-Level Race Detection in STM Haskell using Contracts
Demeyer, Romain; Vanhoof, Wim

Published in:
EPTCS

DOI:
10.4204/EPTCS.137.10

Publication date:
2013

Link to publication
Citation for published version (HARVARD):
Demeyer, R & Vanhoof, W 2013, 'Static Application-Level Race Detection in STM Haskell using Contracts'
EPTCS. https://doi.org/10.4204/EPTCS.137.10

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
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 (TVars) 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, TVars 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 TVars: 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. 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

Footnote 1
Let x be an integer and xs be a list of integers [x₁,...,xₙ], (x:xs) correspond to the list [x,x₁,...,xₙ].
least one access being a write). This is both sufficient and efficient, as long as the values of the TVars 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 TVars 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. We 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.

\[
\text{inc} :: \text{Int} \to \text{Int} \quad \text{-- Type} \\
\text{inc} :: \{ x \mid x > 0 \} \to \{ r \mid r > x \} \quad \text{-- Contract} \\
\text{inc} x = x + 1 \quad \text{-- Definition}
\]

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 BADs are removed during simplification and hence proven to be unreachable), either definitely not safe (the expression does simplify to BAD), or unknown (some BADs
Static Application-Level Race Detection in STM Haskell using Contracts

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.

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 (TVars) in a consistent state. The main idea behind the verification is to deduce contracts from the properties expressing consistency of the TVars, 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 TVars, they can be seen as pure functions taking the values of the TVars 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 TVars to be in a consistent state. This can be done by writing a Haskell function that returns True, respectively False, if the set of TVars are in a consistent, resp. inconsistent, state. In the context of our running example, this function would be as follows:

\[
\text{inv } (\text{shTab,shSum}) = \text{sum shTab == shSum}
\]

\[
\text{where } \text{sum xs = case xs of } [\ ] \rightarrow 0
\]

\[
(1:ls) \rightarrow 1 + \text{sum ls}
\]

Indeed, in our program, the TVars 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 $\mathcal{T}$, 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 $\mathcal{T}(e)$ – i.e. an expression which is completely pure – in such a way that the contracts can be checked on $\mathcal{T}(e)$ 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 $\mathcal{H}$, the language of contracts, and we formally develop the $\mathcal{T}$-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 ($\text{TVar}$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 $\mathcal{H}$, 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 $\mathcal{H}$ 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. Performing I/O operations is not allowed inside transactions.

A first kind of expressions, which we call pure expressions are those constituted by repeated application of only the rules $E_1$ - $E_7$. They correspond to the language defined in [47], being a classical functional language based on construction ($E_2$), lambda abstraction ($E_3$), application ($E_5$), function calls ($E_6$), and case expression ($E_7$). Note the presence of exceptions ($E_4$) – 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 $E_8$ - $E_{11}$. Retrieval of the content of a $\text{TVar}$ ($E_8$), updating the content of a $\text{TVar}$ ($E_9$), binding two STM expressions ($E_{10}$) and defining the return expression ($E_{11}$) 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 $E_8$, $E_9$, $E_{10}$ or $E_{11}$ is of type $\text{STM a}$ where $a$ is the type of the expression returned when evaluating the STM expression, and a $\text{TVar}$ as being of type $\text{TVar a}$ where $a$ is the type of the expression that we can store in this $\text{TVar}$. We will refer to STM expressions of type $\text{STM a}$ 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 $\text{writeTVar t e}$ or $\text{return e}$ appear in an expression, $e$ is a pure expression.

In what follows, we suppose certain types and constructors given. Among them the type $\text{Bool}$ defining the zero-arity constructors $\text{True}$ and $\text{False}$, i.e. $\{\text{True}, \text{False}\} \in \mathcal{H}$. In the examples, we will furthermore use integers and lists, the latter being defined using two constructors: the zero-arity constructor $\text{[]}$ 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
### Function Names ($\mathcal{F}$)

- $f \in \text{Function Names (}\mathcal{F}\text{)}$

### Lambda Variables ($\mathcal{X}$)

- $x, y \in \text{Lambda Variables (}\mathcal{X}\text{)}$

### Data Constructors ($\mathcal{K}$)

- $K \in \text{Data Constructors (}\mathcal{K}\text{)}$

### Transactional Variables ($\mathcal{V}$)

- $t \in \text{Transactional Variables (}\mathcal{V}\text{)}$

### Expressions

- $e, p \in \text{Exp}$

#### Syntax of $\mathcal{H}$ expressions

<table>
<thead>
<tr>
<th>$e, p$</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>variable</td>
</tr>
<tr>
<td>$K \overline{e}$</td>
<td>constructor (value)</td>
</tr>
<tr>
<td>$\lambda x.e$</td>
<td>lambda abstraction (value)</td>
</tr>
<tr>
<td>$r$</td>
<td>exception (value)</td>
</tr>
<tr>
<td>$e_1 e_2$</td>
<td>application</td>
</tr>
<tr>
<td>$f$</td>
<td>function call</td>
</tr>
<tr>
<td>case $e$ of ${alt_1...alt_n}$</td>
<td>case-expression</td>
</tr>
<tr>
<td>readTVar $t$</td>
<td>STM read variable</td>
</tr>
<tr>
<td>writeTVar $t e$</td>
<td>STM write variable</td>
</tr>
<tr>
<td>$e_1 &gt;&gt;= e_2$</td>
<td>STM bind</td>
</tr>
<tr>
<td>return $e$</td>
<td>STM return (value)</td>
</tr>
</tbody>
</table>

#### Exceptions

- $r \in \text{Exceptions}$

- $r ::= \text{BAD}$
  - $\text{UNR}$

#### Alternatives

- $alt ::= K \overline{x} \rightarrow e$

---

**Figure 5:** Syntax of $\mathcal{H}$ expressions

---

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

\[
\begin{align*}
\text{let } x = e' \text{ in } e & \equiv (\lambda x.e) e' \\
\text{do } \{ x <- e; S \} & \equiv e >>= (\lambda x.\text{do } \{ S \}) \\
\text{do } \{ e; S \} & \equiv e >>= (\lambda _.\text{do } \{ S \}) \\
\text{do } \{ e \} & \equiv e
\end{align*}
\]

An expression is evaluated (or reduced) with respect to an environment $\sigma$, which is a mapping from TVars $t_i \in \mathcal{V}$ to pure expressions. We will denote by $\sigma(t_i)$ the (pure) expression that is associated to the TVars $t_i$ in the environment $\sigma$. We will sometimes call this expression the (transactional) value of $t_i$. We will denote by $\sigma[t_i \mapsto e]$ the environment $\sigma'$ 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 $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$ which, from the combination of an expression $e$ and an environment $\sigma$ returns a new expression $e'$ and a new environment $\sigma'$. 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 $\sigma$ 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 $C$ 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
e

\text{reserved for specifying contracts on pure expressions and are identical to those defined in [47]. We call}

\text{C}\)

\text{The syntax of contracts is given in Fig. 7. Contracts defined by application of only the rules}

\text{C1 - C4 are reserved for specifying contracts on pure expressions and are identical to those defined in [47]. We call}

\text{UNR}\)

\text{BAD}\)

\text{BAD}\)

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

\text{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.}

\text{4.2 Contracts}

\text{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}

\text{\langle λx.e_1\rangle e_2, σ → \langle e_1[e_2/x], σ \rangle (APP)}

\text{f = e ∈ pgm \langle f, σ \rangle → \langle e, σ \rangle (CALL)}

\text{\langle \text{case} K_i \overline{σ_i} \text{ of} \{\ldots, K_i \overline{σ_i} \rightarrow e, \ldots\}, \overline{σ} \rangle → \langle e[\overline{e_i}/x_i], \overline{σ} \rangle (CASE)}

\text{\langle \text{readTVar} t, σ \rangle → \langle \text{return} σ(t), σ \rangle (READ)}

\text{\langle \text{writeTVar} t e, σ \rangle → \langle \text{return} (), σ[t \mapsto e] \rangle (WRITE)}

\text{\langle \text{return} e_1 >>= e_2, σ \rangle → \langle e_2 e_1, σ \rangle (BIND)}

\text{\langle e, σ \rangle → \langle e', σ' \rangle (CTX)}

\text{\langle C[\overline{o}[e], σ] \rangle → \langle C[\overline{o}[e'], σ] \rangle (EXC)}

\text{\text{C ::= o | C e_2 | case C of \{alt_1\ldots alt_n\} | C >>= e_2}}

\text{Figure 6: Semantics of C expressions.}
them pure contracts in order to distinguish them from STM contracts which are contracts involving at least one application of the novel rule C5. Intuitively, we will associate pure contracts to pure expressions and STM contracts to STM expressions.

\[
\begin{array}{ll}
\text{Contracts} & \{ x | p \} \quad \text{Predicate Contract} \quad [C1] \\
& x : c_1 \rightarrow c_2 \quad \text{Dependent Function Contract} \quad [C2] \\
& (c_1, c_2) \quad \text{Data Constructor Contract} \quad [C3] \\
& \text{Any} \quad \text{Polymorphic Any Contract} \quad [C4] \\
& \parallel x : c_1 \diamond c_2 \parallel c \quad \text{STM Operation Contract} \quad [C5]
\end{array}
\]

Figure 7: Syntax of contracts

We choose this syntax for the contract C5, 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 C5, 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 \( (a_1, \ldots, a_n) \) where \( a_i \) is the type of the expression stored in the TVars \( t_i \). 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 TVars and returned expressions must be pure expressions.

\[
\forall i : 1 \leq i \leq n : t_i :: \text{TVar } a_1 ; c_1, c_2 :: (a_1, \ldots, a_n) ; c :: a \\
\parallel x : c_1 \diamond c_2 \parallel c :: \text{STM } a
\]

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 CS1 - CS4 are based on the original work from [47]. Intuitively, \( e \in \{ x | p \} \), 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 \( \text{Ok} x \) is just a syntactic notation for a contract \( \{ x | \text{True} \} \). 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 \( c_2 \) 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 \( \mathcal{X} \) with a somewhat nonstandard notation, this contract can effectively be generalized to any constructor from \( \mathcal{X} \). The special contract \( \text{Any} \) is satisfied by any pure expression, including crashing expressions such as Bad.

The definition CS5 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 \( \sigma \) to refer to the pure expression \( (e_1, \ldots, e_n) \) where \( e_i = \sigma(t_i) \) for all \( t_i \in \mathcal{Y} \). 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 TVars, 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 CS1 - CS4. Also note that a diverging expression satisfies
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 \cite{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 TVars 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. $\{(x_1, \ldots, x_n) \mid p\}$ as a shorthand for $\{x \mid \text{case } x \text{ of } (x_1, \ldots, x_n) \rightarrow p\}$, and to avoid repeating variable(s) in (dependent) contracts when it is not useful, i.e. $\| x : \{x \mid p\} \diamond c_2 \| c$ as a shorthand for $\| x : \{x \mid p\} \diamond c_2 \| c$.

\begin{figure}[h]
\centering
\begin{align*}
    e \in \{x \mid p\} & \iff e \text{ is pure and } (e \uparrow^* \text{ or } (e \text{ is crashfree and } p[e/x] \uparrow^* \text{ or } p[e/x] \rightarrow^* \text{True})) \quad [\text{CS1}] \\
    e \in x : c_1 \rightarrow c_2 & \iff 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]) \quad [\text{CS2}] \\
    e \in (c_1, c_2) & \iff e \uparrow^* \text{ or } (e \rightarrow^* (e_1, e_2) \text{ and } e_1 \in c_1, e_2 \in c_2) \quad [\text{CS3}] \\
    e \in \text{Any} & \iff e \text{ is pure.} \quad [\text{CS4}] \\
    e \in \| x : c_1 \diamond c_2 \| c & \iff \forall \sigma : \overrightarrow{\text{c}} \in c_1 : \langle e, \sigma \rangle \uparrow^* \text{ or } (\langle e, \sigma \rangle \rightarrow^* \langle \text{return } e', \sigma' \rangle \text{ and } \overrightarrow{\sigma'} \in c_2[\overrightarrow{\text{c}}/x] \text{ and } e' \in c[\overrightarrow{\text{c}}/x]) \quad [\text{CS5}]
\end{align*}
\caption{Contract Satisfaction}
\end{figure}

\begin{figure}[h]
\begin{verbatim}
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
\end{verbatim}
\caption{The function send and its type and contract.}
\end{figure}

Note that our extensions to the contract system of \cite{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 $\lambda g.e \in c_g \rightarrow c_f$ 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 $e \in \parallel c \diamond c \parallel \forall x$. 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

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

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 TVars, 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 $e \in \parallel c \diamond c \parallel \forall x$, one must verify whether $(\lambda x_1...\lambda x_n.e) \in c_1 \times \cdots \times c_n \rightarrow \parallel c \diamond c \parallel \forall x$, 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 0k 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 such a way that $e' \in c'$ implies $e \in c$. This transformation, represented by the $\mathcal{T}$-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. $\mathcal{V} = \{t\}$, but the definitions can readily be extended towards handling a given set of TVars.

The basic idea is to transform a STM operation $e$ returning $e'$ and possibly updating the (transactional) value of $t$ into a lambda expression $\lambda t.(e', e'\prime)$ where $e'\prime$ is the updated value of $t$. Reading the TVar is modeled by a function that associates to the initial value of $t$ the couple $(i, t)$. 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 0, 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
\[ T(\text{readTVar } t) = \lambda t. (t, t) \]
\[ T(\text{writeTVar } t e) = \lambda t. (t, e) \]
\[ T(e_1 >> e_2) = \lambda t. (T(e_2) (\text{fst } e'_1) \text{ (snd } e'_1)) \quad \text{where } e'_1 = T(e_1) t \\
\text{fst} = \lambda (a, b). a \\
\text{snd} = \lambda (a, b). b \]
\[ T(\text{return } e) = \lambda t. (e, t) \]
\[ T(f) = \lambda t. (f t) \quad \text{if } f :: \text{STM } a. \]
\[ = f \quad \text{else.} \]
\[ T(e_1 e_2) = \lambda t. (T(e_1) T(e_2) t) \quad \text{if } e_1 e_2 :: \text{STM } a. \]
\[ = T(e_1) T(e_2) \quad \text{else.} \]
\[ T(\lambda x.e) = \lambda x. T(e) \]
\[ T(K e) = K T(e) \]
\[ T(x) = x \]
\[ T(r) = r \]
\[ T(\text{case } e \text{ of } \text{pat}_i \to T(e_i)) = \lambda t. ((\text{case } T(e) \text{ of } \text{pat}_i \to T(e_i)) t) \quad \text{if } e \text{ case } :: \text{STM } a. \]
\[ = \text{case } T(e) \text{ of } \text{pat}_i \to T(e_i) \quad \text{else.} \]

where \( e \text{ case } \equiv \text{ case } e \text{ of } \text{pat}_i \to e_i \)

Figure 11: \( T \)-transformation for \( \mathcal{H} \) expressions

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

\[
T(\text{readTVar } t >>= \lambda x.\text{writeTVar } t (x+1)) = \lambda t. ((\lambda x. (\lambda t. ((), x+1))) (\text{fst } ((\lambda t. (t)) t)) \text{ (snd } ((\lambda t. (t)) t) t))
\]

which can symbolically [22] be rewritten into \( \lambda t. (((), t+1) \to \lambda t. (t)) t \). 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 \( \text{TVar} \).

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 \( T \)-operator is defined such that:

- if \( e \) is a STM operation, then the execution of \( T(e) \ \text{ e}' \) 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 \( \text{TVar} \) if the latter would have contained \( e' \) before performing the STM operation.

- if \( e \) is a pure expression, then the execution of \( T(e) \) will produce the same result as the execution of \( e \).

Moreover, the \( T \)-operator can be generalized easily for a (totally ordered) set of \( n \) \( \text{TVars} \), i.e. \( \mathcal{V} \equiv \{t_1, \ldots, t_n\} \). This would require transforming into an expression that takes as argument a tuple \( (t_1, \ldots, t_n) \) 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:

\[
T(\text{readTVar } t_k) = \lambda (t_1, \ldots, t_k, \ldots, t_n). (t_k, (t_1, \ldots, t_k, \ldots, t_n))
\]

\[ ^2\text{we use the following syntactic sugar: } \lambda (t_1, \ldots, t_n). e \equiv \lambda x. \text{case } x \text{ of } (t_1, \ldots, t_n) \to e \text{ where } x \text{ is not a free variable of } e. \]
We also need a technique to convert STM contracts into pure contracts. For this purpose, we override the \( \mathcal{T} \)-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.

\[
\mathcal{T}(\{ x \mid p \}) = \{ x \mid p \} \\
\mathcal{T}(x : c_1 \to c_2) = x : \mathcal{T}(c_1) \to \mathcal{T}(c_2) \\
\mathcal{T}(c_1, c_2) = (\mathcal{T}(c_1), \mathcal{T}(c_2)) \\
\mathcal{T}(\text{Any}) = \text{Any} \\
\mathcal{T}(\langle x : c_1 \diamond c_2 \parallel c \rangle) = x : c_1 \to \langle c, c_2 \rangle
\]

Fig. 12: \( \mathcal{T} \)-transformation for contracts

Fig. 13 enlists a number of easily proven but important properties of the \( \mathcal{T} \)-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 \( \mathcal{T} \) and the fact that we consider only well-formed contracts, i.e. subcontracts of STM contracts are pure. Secondly, \( \mathcal{T} \) 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 \( \text{BAD} \) and \( \text{UNR} \) 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.

\begin{itemize}
  \item \( \mathcal{T}(e) \), resp. \( \mathcal{T}(c) \), is a pure expression, resp. contract. \hspace{1cm} (1)
  \item \( \mathcal{T}(\mathcal{T}(e)) \equiv \mathcal{T}(e), \mathcal{T}(\mathcal{T}(c)) \equiv \mathcal{T}(c) \hspace{1cm} (2) \)
  \item \( \mathcal{T}(e) \equiv e \) if \( e \) is a pure expression. \hspace{1cm} (3)
  \item \( \mathcal{T}(c) \equiv c \) if \( c \) is a pure contract. \hspace{1cm} (4)
  \item \( \mathcal{T}(e) \to^* \lambda x. e' \) if \( e :: \text{STM a} \) and \( e \neq r \). \hspace{1cm} (5)
  \item \( \langle e, \sigma_1 \rangle \to^* \mathcal{T}(e) \sigma_1 \to^* \mathcal{T}(e) \sigma_2 \) if \( e :: \text{STM a} \hspace{1cm} (6) \)
  \item \( \langle e, \sigma_1 \rangle \to^* \mathcal{T}(e) \sigma_1 \to^* \mathcal{T}(e) \sigma_2 \) \hspace{1cm} (7)
\end{itemize}

Figure 13: Properties of \( \mathcal{T} \)

Trivially, as a direct consequence of properties (3) and (4), we have that \( e \in c \iff \mathcal{T}(e) \in \mathcal{T}(c) \) 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 \( a \),

\[
e \in c \iff \mathcal{T}(e) \in \mathcal{T}(c)
\]

**Proof.** As \( c \) is of type STM \( a \), it is a contract of the form \( c \equiv \| c_1 \diamond c_2 \parallel c' \). We distinguish three cases:

\begin{itemize}
  \item \( e \equiv \text{BAD} \) : We can see, by the semantics of contracts and expressions, that \( \text{BAD} \notin \| c_1 \diamond c_2 \parallel c' \) and \( \mathcal{T}((\text{BAD})) \equiv \text{BAD} \notin x : c_1 \to (c', c_2) \equiv \mathcal{T}((\| c_1 \diamond c_2 \parallel c')) \) for any \( c_1, c_2, c' \).
  \item \( e \equiv \text{UNR} \) : Similarly, \( \text{UNR} \in \| c_1 \diamond c_2 \parallel c' \) and \( \mathcal{T}((\text{UNR})) \equiv \text{UNR} \in x : c_1 \to (c', c_2) \equiv \mathcal{T}((\| c_1 \diamond c_2 \parallel c')) \) for any \( c_1, c_2, c' \).
\end{itemize}
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,

\[ e \in c \iff T(e) \in T(c) \]

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 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.

### 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`\[^3\]. 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.

\[
\text{atomically ( do } c \leftarrow \text{readTVar connected}
\quad \text{case } c \text{ of True } \rightarrow \text{sendMessage}
\quad \text{False } \rightarrow \text{retry })
\]

There is no difficulty to deal with retry in our framework. It is sufficient to define \( T(\text{retry}) = \text{UNR} \).
Indeed, recall that we do not have to deal with diverging expressions, nor with the values of the
TVars at the intermediate states of the transaction, but only with the possible values of the TVars 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, \( e_1 \text{ `orElse` } e_2 \) proceeds as follows: first, \( e_1 \) is evaluated. If its evaluation does \textit{not} result in retry,
the operation ends with the result computed by \( e_1 \). If on the other hand, \( e_1 \) \textit{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 ver-
sions of the transaction, say \( \Gamma(e) \), one version for each possible combination of alternative evaluations.
The \( \Gamma \)-operator computing all possible such evaluations is partly defined below:

\[
\begin{align*}
\Gamma(e_1 \text{ `orElse` } e_2) &= \Gamma(e_1) \cup \Gamma(e_2) \\
\Gamma(e_1 >>= e_2) &= \{ e'_1 >>= e'_2 \mid e'_1 \in \Gamma(e_1), e'_2 \in \Gamma(e_2) \} \\
\Gamma(\lambda x. e) &= \{ \lambda x. e' \mid e' \in \Gamma(e) \}
\end{align*}
\]

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

\section*{TVars as arguments of a function}

One limitation of our language \( H \) is that, contrary to full STM Haskell, it does \textit{not} allow a function
(or, more precisely, a lambda abstraction) to have TVars as its arguments. However, as long as we
consider a fixed set of TVars manipulated by the program, this limitation can be overcome by using
well-known techniques from program specialization \cite{21,19} in order to specialize both the function and
its corresponding contract with respect to all possible subsets of TVars 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.

\[
f :: \text{TVar Int} \rightarrow \text{STM ()}
f x = \text{do } n \leftarrow \text{readTVar} x
\quad \text{writeTVar} x (n+1)
\]

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:
Supposing now that the program manipulates two TVar s, say \( Y = \{ tA, tB \} \), 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):

\[
\begin{align*}
f \_tA :: & \mid \{ (tA,tB) \mid tA \geq 0 \} <> \{ (tA',tB') \mid tA' \geq tA \} \mid Any \\
& | Any \\
f \_tA = & \text{ do } n <- \text{ readTVar } tA \\
& \text{ writeTVar } tA (n+1) 
\end{align*}
\]

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 [11]. 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


