Retriable Futures

Martin Sulzmann* and Jonas Moosmann**

Hochschule Karlsruhe - Technik und Wirtschaft

Abstract. Futures are a programming language abstraction to structure asynchronous computations. We argue that there are several situations where we wish that a computation associated to a future will be retried, e.g. in case of failure or if some condition is not satisfied. We formalize the concept of retriable futures and show how to implement them by using a standard future API combined with some standard Software Transactional Memory (STM) operations. We have implemented the approach in Haskell and Scala and provide some experimental results.

1 Introduction

A future [2] serves as a place-holder for a read-only variable whose result is as yet undetermined. The computation producing the result is carried out asynchronously in a separate thread. In the main thread, we can either (a) wait until the 'future' computation has been completed, or (b) define a callback functions via which we can query the 'future' result. The advantage of (b) is that we will not block the main thread.

Examples Consider the following program written in some functional style language with support for futures.

$$
\begin{array}{l}
x_1 = \mathsf{future}(\mathit{rateEuroToDollar}(\mathsf{100})); \\
\mathsf{success}(\mathsf{x_1}, \lambda \mathsf{r}.\mathit{tellRate}(\mathsf{r})); \\
\mathsf{failure}(\mathsf{x_1}, \mathit{checkWhatWentWrong}); \\
...//\text{do something else}
\end{array}
$$

We define a future x_1 to obtain some exchange rate (Euro→ Dollar). The function call $\mathit{rateEuroToDollar}(100)$ consults several banks and will select the best rate which is then assigned to x_1.

Instead of blocking the main thread and waiting for the rate to become available, we process the outcome of the exchange rate computation via some callback functions. In case of success, the non-blocking statement success applies. In case of failure, e.g. due to network problems, failure applies.

Besides future(), practical implementations [11, 6] of futures typically support some rich set of operators to compose futures. Thus, we can model more complex scenarios as the following example shows.

* martin.sulzmann@hs-karlsruhe.de
** j.moosmann@gmx.de

Operators Suppose, we are planning some holiday where based on some suitable exchange rate we wish to perform some (hotel/airline) booking. In terms of the next operator, this can be expressed as follows.

$$x_1 = \mathsf{future}(\mathit{rateEuroToDollar}(100));$$
$$x_2 = \mathsf{next}(\mathsf{x}_1, \lambda\mathsf{x}.\mathit{bookUS}(\mathsf{x}));$$

Operator next effectively waits for the result of x_1 to become available. The result (if available) will be passed to *bookUS* which creates a new future x_2. If x_1 fails, x_2 will fail. Importantly, operators such as next are executed asynchronously and will therefore not block the main thread.

Let's say we are considering two alternatives, either a holiday to the US or Switzerland. We introduce yet another operator alt to select among alternative bookings.

$$x_1 = \mathsf{future}(\mathit{rateEuroToDollar}(100));$$
$$x_2 = \mathsf{next}(\mathsf{x}_1, \lambda\mathsf{x}.\mathit{bookUS}(\mathsf{x}));$$
$$x_3 = \mathsf{future}(\mathit{rateEuroToFranc}(100));$$
$$x_4 = \mathsf{next}(\mathsf{x}_3, \lambda\mathsf{x}.\mathit{bookSwiss}(\mathsf{x}));$$
$$x_5 = \mathsf{alt}(\mathsf{x}_3, \mathsf{x}_4);$$
$$\mathsf{success}(\mathsf{x}_5, \lambda\mathsf{r}.\mathit{hoorayHoliday}(\mathsf{r}));$$

Operator alt chooses among two futures where preference is given to the first alternative. That is, if the first computation fails, the second future is considered. Obviously, it make sense to consider both options x_2 and x_4 concurrently, hence, the use of futures.

As we can see, this style of writing concurrent programs turns out to be quite useful in practice. It often requires only small changes to existing sequential programs and generally improves the program's efficiency assuming that concurrent computations can be executed in parallel.

Failure What if x_2 and x_4 fail? For example, the booking system may be temporarily down and therefore the US as well as the Switzerland booking will fail. One possibility is to catch failure of x_5 and simply repeat *all* of the earlier steps. This will be wasteful in case we have already obtained the exchange rates x_1 and x_3.

Instead of catching only failure of the final future, we could insert subchecks and memorize subresults. Ultimately, we will need to consider all (failure) possibilities which leads to some complicated control flow which must be managed manually by the programmer. This is error prone and leads to non-composable program code which becomes much harder to read.

Retriable Futures Our idea is to introduce a new operator retry which retries a future until the future finally succeeds. Below, we apply this extension to our running example where for brevity we omit some of the definitions.

```
...
x5 = retry(alt(x3, x4));                    //RETRY
x6 = any(x5, future(timeout(1h)))
success(x6, λr.hoorayHoliday(r));
failure(x6, alternativePlan);
```

If say both booking options yield failure, the retry mechanism will automatically restart the necessary subcomputations.

Of course, we could experience permanent failure (or no result at all). To avoid some indefinite retry, we simply impose a time limit on the number of retries by introducing x_6. The computation $\mathsf{future}(\mathit{timeout}(\mathit{1h}))$ yields success after about one hour and the operator any chooses (in-deterministically) the first available successful result.

A retry may be necessary not only in case of failure but also if some condition is not satisfied. For example, consider the following example.

```
x1 = future(getQuote);
x2 = retry(x1, λr.r >= 40);
```

The future x_1 retrieves some stock exchange information which we will only accept if the quote satisfies a certain condition. Otherwise, we will issue a retry.

In summary, we propose a retry mechanism for futures where the computations associated to a future may be retried. A retry may take place in case of failure or if some condition is not satisfied. The retry mechanism for futures changes their semantics. Standard futures are read-only whereas the result of a retriable future may be overwritten. The above examples show typical use cases for such a feature. The advantage of a retriable future for the programmer is that the program code remains (almost) the same. The burden is put on the implementation of retry to ensure that only the necessary subcomputations are retried without having to retry the entire computation.

Contributions and Outline Specifically, we make the following contributions:

- We formalize the semantics of retriable futures (Section 2).
- We show how to implement retriable futures by by adding an STM-based retry management layer to an existing set of standard future operations (Section 3).
- We conduct some experiments to measure the overhead of our retriable future operations compared to a standard operations (Section 3.4).

Section 4 discusses related work and concludes.

2 Retriable Futures

We formalize a minimal language with support for retriable futures.

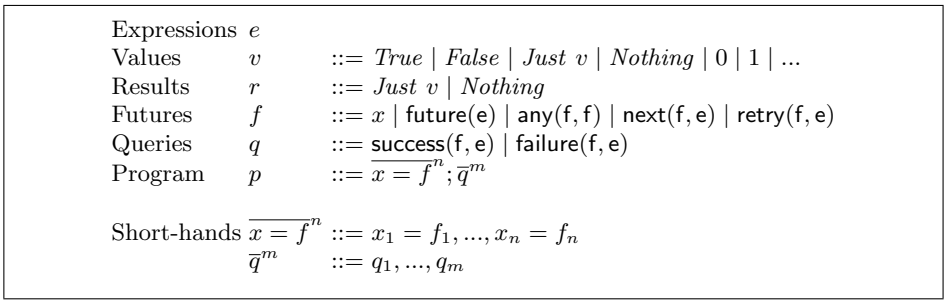

Fig. 1: Syntax

2.1 Language Syntax

The syntax f of futures is defined in Figure 1. A future f yields either a successful result of fails if the computation terminates at all. The failure case describes exceptional program behavior, e.g. division by zero, timeout caused by unavailable web service etc. For simplicity, we represent both outcomes in terms of the *Maybe* data type which has the two constructor values *Just* v and *Nothing*. The former represents the case that the computation has succeeded with value v and the latter represents the case that the computation has failed. A more realistic implementation would for example represent the failure case by raising some exception.

Our language supports some basic operators to build futures. The form $\mathsf{future(e)}$ specifies a primitive (future) computations where the actual computation is specified in terms of some host language expression e. For brevity, we will largely ignore the details of the host language as the specifics do not matter for the formalization of retriable futures.

The form $\mathsf{alt(f_1, f_2)}$ represents a choice between two future computations where preference is given to f_1. The form $\mathsf{next(f, e)}$ passes the result of f to function e which then results in a new future. For brevity, we omit further operators such as $\mathsf{any(f_1, f_2)}$ (indeterministic choice) in our formal development. Our implementation supports such features.

The new and interesting bit is the form $\mathsf{retry(f, e)}$ which issues a retry of f until boolean condition specified by the predicate expression e is satisfied. The form $\mathsf{retry(f)}$, which issues a retry as long as f fails to produce a successful result, is a short-hand for $\mathsf{retry(f, \lambda\_.}\mathit{True}\mathsf{)}$.

To query the outcome of a future computation we use the forms $\mathsf{success(f, e)}$ and $\mathsf{failure(f, e)}$ The form $\mathsf{success(f, e)}$ applies the (callback) function e on the successful result of future f. The form $\mathsf{failure(f, e)}$ calls e if the computation has failed. A program consists of a sequence of future definitions and queries.

Next, we will formalize the semantics of retriable futures. Before diving into formal details, we highlight some of the technical aspects via some examples.

2.2 Operational Semantics Overview

Consider the following set of definitions where the details of host expressions e_1 and e_2 are left unspecified.

$$\begin{aligned} x_1 &= \mathsf{future}(\mathsf{e_1});\\ x_2 &= \mathsf{future}(\mathsf{e_2});\\ x_3 &= \mathsf{retry}(\mathsf{alt}(\mathsf{x_1}, \mathsf{x_2}), \lambda\_.\mathit{True});\end{aligned}$$

Suppose the results of futures x_1 and x_2 are available and recorded in some state environment S where $S = \{x_1 \mapsto \mathit{Nothing}, x_2 \mapsto \mathit{Nothing}\}$. That is, x_1 and x_2 have failed. Next, we proceed to evaluate x_3. Evaluation of subexpression $\mathsf{alt}(\mathsf{x_1}, \mathsf{x_2})$ fails. Hence, we issue a retry which effectively implies that we need to re-evaluate x_1 and x_2 which for example yields the updated state $S' = \{x_1 \mapsto \mathit{Nothing}, x_2 \mapsto \mathit{Just}\ 2\}$.

The observation is that during the evaluation of expressions, we need to keep track of the set of to be retried definitions. Besides the result of a future expression, we also obtain the update state of all retried definitions.

More formally, for the semantic description we will employ judgments

$$S, \overline{x = f}^{\,n} \vdash_R f \Downarrow (S, r)$$

where S, $\overline{x = f}^{\,n}$, R and f to the left of $\Downarrow$ are input parameters whereas the pair (S, r) is the output. The input state S contains the results of futures whose computation has already taken place. In addition, we also find the sequence of future definitions $\overline{x = f}^{\,n}$ in a program. The parameter R controls which futures will be retried. R is assumed to be initially empty and will only be adjusted in case of a retry. The output (S, r) contains besides the result r the collected results of all futures which have been retried. To formalize the semantic rules we employ a big-step semantics.

2.3 Formal Details

Futures The semantic rules for futures are given Figure 2. In rule (x1) we retry the definition f associated to x. Evaluation of f may incur further retries which are collected in S'. Hence, we find $S' \cup \{x \mapsto r\}$ as the state of updated results of retried futures. Rule (x2) covers the case where no retry shall take place and we simply look up the result in S. Rule (F) deals with a primitive future where we assume that $\Downarrow_H$ refers to the semantic evaluation function of the host language. The details of $\Downarrow_H$ as well as the concrete syntax of host expressions are left out for simplicity.

Rules (A1-2) cover evaluation of $\mathsf{any}(\mathsf{f_1}, \mathsf{f_2})$. Recall that we shall give preference to f_1. If successful, we therefore do not need to consider f_2. See rule (A1). Otherwise, we evaluate f_2. See rule (A2). In the combined output $S_1 \cup S_2$, we might find some clashes if some x occurring in f_1 and f_2 will be retried in both branches. We resolve such clashes by giving preference to the retries from the right operand by aggressively applying the following rule (from left to right).

$$S \cup \{x \mapsto r\} \cup S' \cup \{x \mapsto r'\} \cup S' = S \cup S' \cup \{x \mapsto r'\} \cup S' \tag{1}$$

State $S ::= \{x_1 \mapsto r_1, ..., x_n \mapsto r_n\}$
Retriable futures $R := \{x_1, ..., x_m\}$

Set of free (retriable) definitions where d is a short-hand for $\overline{x = f}^n$:

$$fv_d(\mathsf{future(e)}) = \{\} \quad fv_{\mathsf{d}}(\mathsf{alt(f_1, f_2)}) = fv_{\mathsf{d}}(\mathsf{f_1}) \cup fv_{\mathsf{d}}(\mathsf{f_2}) \quad fv_{\mathsf{d}}(\mathsf{next(f, e)}) = fv_{\mathsf{d}}(\mathsf{f})$$
$$fv_d(\mathsf{retry(f, e)}) = \{\} \quad fv_{\overline{\mathsf{x=f}}^{\mathsf{n}}}(\mathsf{x}) = \begin{cases} \{x\} \cup fv_{\overline{x=f}^n}(f_i) & \text{if } x = x_i \text{ for some } i \\ \{x\} & \text{otherwise} \end{cases}$$

$$\boxed{S, \overline{x = f}^n \vdash_R f \Downarrow (S, r)}$$

$$\text{(x1)} \ \frac{x \in R \quad x = f \in \overline{x = f}^n \quad S, \overline{x = f}^n \vdash_R f \Downarrow (S', r) \quad S'' = S' \cup \{x \mapsto r\}}{S, \overline{x = f}^n \vdash_R x \Downarrow (S'', r)} \qquad \text{(x2)} \ \frac{x \notin R \quad S(x) = r}{S, \overline{x = f}^n \vdash_R x \Downarrow (\{\}, r)}$$

$$\text{(F)} \ \frac{e() \Downarrow_H r}{S, \overline{x = f}^n \vdash_R \mathsf{future(e)} \Downarrow (\{\}, \mathsf{r})}$$

$$\text{(A1)} \ \frac{S, \overline{x = f}^n \vdash_R f_1 \Downarrow (S_1, \mathit{Just}\ v)}{S, \overline{x = f}^n \vdash_R \mathsf{alt(f_1, f_2)} \Downarrow (\mathsf{S_1}, \mathit{Just}\ \mathsf{v})}$$

$$\text{(A2)} \ \frac{S, \overline{x = f}^n \vdash_R f_1 \Downarrow (S_1, \mathit{Nothing}) \quad S, \overline{x = f}^n \vdash_R f_2 \Downarrow (S_2, r)}{S, \overline{x = f}^n \vdash_R \mathsf{alt(f_1, f_2)} \Downarrow (\mathsf{S_1} \cup \mathsf{S_2}, \mathit{Just}\ \mathsf{v})}$$

$$\text{(N1)} \ \frac{S, \overline{x = f}^n \vdash_R f \Downarrow (\mathit{Nothing}, S')}{S, \overline{x = f}^n \vdash_R \mathsf{next(f, e)} \Downarrow (\mathsf{S'}, \mathit{Nothing})} \qquad \text{(N2)} \ \frac{S, \overline{x = f}^n \vdash_R f \Downarrow (\mathit{Just}\ v, S') \quad e(v) \Downarrow_H r}{S, \overline{x = f}^n \vdash_R \mathsf{next(f, e)} \Downarrow (\mathsf{S'}, \mathsf{r})}$$

$$\text{(R1)} \ \frac{S, \overline{x = f}^n \vdash_R f \Downarrow (\mathit{Just}\ v, S') \quad e(v) \Downarrow_H \mathit{True}}{S, \overline{x = f}^n \vdash_R \mathsf{retry(f, e)} \Downarrow (\mathit{Just}\ \mathsf{v}, \mathsf{S'})}$$

$$\text{(R2)} \ \frac{S, \overline{x = f}^n \vdash_R f \Downarrow (\mathit{Nothing}, \_) \quad R' = fv_{\overline{x=f}^n}(f) \quad \{\}, \overline{x = f}^n \vdash_{R'} \mathsf{retry(f, e)} \Downarrow (\mathit{Just}\ \mathsf{v}, \mathsf{S'})}{S, \overline{x = f}^n \vdash_R \mathsf{retry(f, e)} \Downarrow (\mathit{Just}\ \mathsf{v}, \mathsf{S'})}$$

$$\text{(R3)} \ \frac{S, \overline{x = f}^n \vdash_R f \Downarrow (\mathit{Just}\ v, \_) \quad e(v) \Downarrow_H \mathit{False} \quad R' = fv_{\overline{x=f}^n}(f) \quad \{\}, \overline{x = f}^n \vdash_{R'} \mathsf{retry(f, e)} \Downarrow (\mathit{Just}\ \mathsf{v}, \mathsf{S'})}{S, \overline{x = f}^n \vdash_R \mathsf{retry(f, e)} \Downarrow (\mathit{Just}\ \mathsf{v}, \mathsf{S'})}$$

Fig. 2: Operational Semantics - Futures

Rules (N1-2) cover $\mathsf{next(f, e)}$ where rule (N1) is responsible for the failure case and rule (N2) for the successful case. We assume that e does not depend on any future definitions and simply evaluate $e(v)$ in terms of the host language.

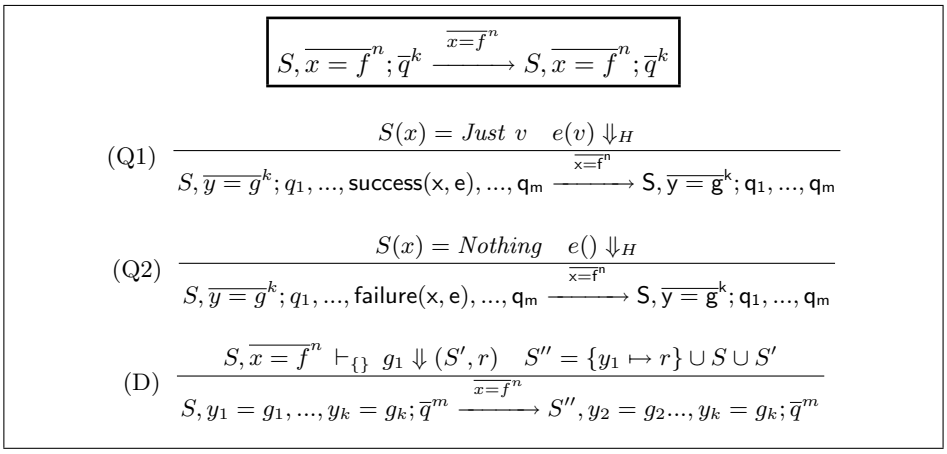

Fig. 3: Operational Semantics - Definitions and Queries

Rules (R1-3) deal with $\mathsf{retry(f, e)}$. We possibly may need to retry the definitions in f. The to be retried definitions in f are computed via $fv_{\overline{x=t}^n}(f)$. See Figure 2. There is no need to collect definitions 'below' another nested retry statement as those conditions will be dealt with by the nested retry statement.

Rule (R1) covers the successful case. In rule (R2), we issue a retry due to failure. Future $\mathsf{retry(f, e)}$ will be retried under the empty state and the set R' of to be retried definitions which are defined in $\overline{x = f}^n$. Rule (R3) covers the successful where the condition could not be satisfied. Hence, we issue another retry.

Definitions and Queries What remains is to specify the semantics of definitions and queries. Definitions $\overline{x = f}^n$ and $\overline{q}^k$ are evaluated in terms of a rewrite relation expressed by the judgment $S, \overline{x = f}^n; \overline{q}^k \xrightarrow{\overline{x=f}^n} S, \overline{x = f}^n; \overline{q}^k$. See Figure 3.

The rewrite relation operates on triples $S, \overline{x = f}^n; \overline{q}^k$. On the left-hand side, state S contains the results of futures whose computation has already taken place and $\overline{x = f}^n; \overline{q}^k$ are the yet to be processed definitions and queries. On the right-hand side, we find the remaining definitions and queries and the updated state.

In our setting, definitions may be retried for which we require the entire set of definitions available in a program. We record this 'global' set of definitions above the transition arrow, written $\xrightarrow{\overline{x=f}^n}$. To avoid confusion, in the specific rules we write $\overline{y = g}^k$ to refer to the processed definitions and $\overline{x = f}^n$ to refer to the set of all definitions available in a program.

Queries are processed in-deterministically by selecting any query for which the respective result is available. For example, consider rule (Q1) which covers the successful query case. We perform a look-up on state S and assume that a successful result v is available. Then, the callback function e is applied on v.

Recall that $\Downarrow_H$ denotes evaluation of host expressions and the concrete rules are left out for simplicity. Similarly, rule (Q2) covers the failed query case.

To perform a query, the result must be available. The actual evaluation of definitions is carried out by rule (D). Definitions will be processed in a strict sequence. Hence, we assume that bindings $x_1 = f_1, ..., x_n = f_n$ are ordered such that x_{i+1} does not appear in any of the earlier expressions f_1, ..., f_i.

The right-hand side of a definition $x_1 = g_1$ is evaluated under state S and the set $\overline{x = f}^n$ of programs definitions. These definitions are necessary because we may retry definitions within g_1. Evaluation is carried out by judgment $S, \overline{x = f}^n \vdash_{\{\}} g_1 \Downarrow (S', r)$ which yields some result r and some S' which contains the results of all retried definitions. In the resulting state S'', S' shall overwrite any earlier result recorded in S by applying the earlier defined simplification rule (1) on states.

2.4 Discussion

Retriable future programs may be indeterministic as shown by the following variant of an example from the introduction.

$$
\begin{array}{l}
x_1 = \mathsf{future}(\mathit{rateEuroToDollar}(100)); \\
x_2 = \mathsf{retry}(\mathsf{x}_2, \lambda \mathsf{r}.\mathsf{r} > 40); \\
\mathsf{success}(\mathsf{x}_1, \mathsf{e}_1); \\
\mathsf{success}(\mathsf{x}_2, \mathsf{e}_2);
\end{array}
$$

Both success statements will process any available successful exchange rate. Due to retry we will only ever pass rates above 40 to e_2. On the other hand, it is entirely possible that e_1 will receive some rate below 40.

For example, if x_1 yields initially 30 we can immediately process $\mathsf{success}(\mathsf{x}_1, \mathsf{e}_1)$. Suppose after some retry of x_1 we obtain 50 which only then allows us to process $\mathsf{success}(\mathsf{x}_2, \mathsf{e}_2)$. Obviously, we could also process $\mathsf{success}(\mathsf{x}_1, \mathsf{e}_1)$ after the retry. Due avoid such indeterministic behavior we could reject programs which query futures which appear within a retry statement.

Another point worth mentioning concerns possible optimizations. Our current strategy is to retry *all* definitions. This may be unnecessary in case we extend our language with some further operators such as $\mathsf{combine}(\mathsf{f}_1, \mathsf{f}_2, \mathsf{e})$. We assume that function e processes the successful results of f_1 and f_2. If f_1 fails the operator call fails as well. The corresponding semantic rules are as follows.

$$
\frac{S, \overline{x = f}^n \vdash_R f_1 \Downarrow (\mathit{Just}\ v_1, S') \quad S, \overline{x = f}^n \vdash_R f_1 \Downarrow (\mathit{Just}\ v_2, S'') \quad e(v_1, v_2) \Downarrow_H v}{S, \overline{x = f}^n \vdash_R \mathsf{combine}(\mathsf{f}_1, \mathsf{f}_2, \mathsf{e}) \Downarrow (\mathit{Just}\ \mathsf{v}, \mathsf{S}' \cup \mathsf{S}'')}
$$

$$
\frac{S, \overline{x = f}^n \vdash_R f_1 \Downarrow (\mathit{Nothing}, S')}{S, \overline{x = f}^n \vdash_R \mathsf{combine}(\mathsf{f}_1, \mathsf{f}_2, \mathsf{e}) \Downarrow (\mathit{Nothing}, \mathsf{S}')}
$$

Let's consider the following program making use of the extended language.

$$x_1 = \mathsf{future}(\mathit{rateEuroToDollar}(100));$$
$$x_2 = \mathsf{future}(\mathit{rateEuroToFranc}(100));$$
$$x_3 = \mathsf{retry}(\mathsf{combine}(\mathsf{x}_1, \mathsf{x}_2, \lambda \mathsf{r}_1.\lambda \mathsf{r}_2.\mathsf{r}_1 + \mathsf{r}_2)$$

We wish to obtain some successful exchange rate in Dollar and Franc where the combination function simply sums up both rates. Suppose x_1 fails but x_2 succeeds. In our current formulation we will retry x_1 *and* x_2. This seems unnecessary here because for x_3 to succeed it would be sufficient to retry x_1 only.

A possible improvement would be to distinguish between retries due to plain failure and retries due to violation of some condition. In case of a retry due to plain failure, there is no need to retry an already successful future. We could track both cases by employing a type and effect discipline [12]. The type $\mathit{RFuture}(t)^{\delta}$ of a future is enriched with an effect annotation δ where δ ranges over $\neg$ and $\_$. The type $\mathit{RFuture}(t)^{\neg}$ denotes a future of type t which may only be retried in case of failure whereas $\mathit{RFuture}(t)^{\_}$ denotes a future which may be retried regardless of any prior result.

The type of retry statements is as follows.

$$\mathsf{retry} :: \forall \mathsf{a}.\mathit{RFuture}(a)^{\_} \rightarrow (\mathsf{a} \rightarrow \mathit{Bool}) \rightarrow \mathit{RFuture}(a)^{\_}$$
$$\mathsf{retry} :: \forall \mathsf{a}, \delta.\mathit{RFuture}(a)^{\delta} \rightarrow \mathit{RFuture}(a)^{\delta}$$

The unconditional retry statement covers both cases because such statements may appear within a conditional retry statement. Thus, we find for our running example that x_3 has type $\mathit{RFuture}(t)^{\neg}$. This information can be exploited by the run-time system to retry only failed futures.

3 Implementation

We report on a prototype implementation of retriable futures. Our prototype covers the full language spectrum but does not yet incorporate any optimizations (see the above discussion). Importantly, we show how to implement retriable futures in terms of an existing future API. Thus, taking advantage of existing highly-tuned existing implementations.

Our idea is to add a retry management layer to an existing (standard) future API. The management layer is responsible to issue retries and maintain the results of retriable futures. Furthermore, each standard future API operation is wrapped with some control information and thus becomes a retriable operation.

To implement the above idea we make use of Software Transactional Memory (STM) operations. Atomic execution guarantees that we maintain a consistent view of retriable futures. The STM retry mechanism allows us to suspend a transaction until some transactional variables have changed. Thus, we obtain a fully workable implementation of retriable futures in a few lines of code.

In the following, we show the *complete* implementation of our idea in Haskell. Figure 4 summarizes the main functionality of an STM and Future API required to implement retriable futures. [1] We will explain the details of the STM and

[1] There exists no designated Future API in Haskell. We simply (re)model the Scala Future API in Haskell terms.

Future API when necessary. We also have a working implementation in Scala which is an almost literal adaptation of the Haskell implementation (but slightly longer).

```
data STM a
data TVar a
newTVarIO  :: a → IO (TVar a)
newTVar :: a → STM (TVar s)
readTVar :: TVar a → STM a
writeTVar :: TVar a → a → STM ()
atomically :: STM a → IO a
retry      :: STM a

data Future a
future :: IO (Maybe a) → Future a
success :: (a → IO ()) → Future a → IO ()
failure :: IO () → Future a → IO ()
alt :: Future a → Future a → IO (Future a)
...
data Promise a
trySuccess :: Promise a → a → IO Bool
tryFail :: Promise a → IO ()
```

Fig. 4: Haskell STM/Future API

3.1 Retriable Futures

First, we consider the definition of retriable futures and how to query success. See Figure 5 for details. To clearly distinguish between standard and retriable operations, we will add the suffix `R` to all retriable operations.

A retriable future (`RFuture`) is represented as a pair of two transactional variables. The first variable describes the result which can either be undetermined ("empty"), failed or successful and containing some actual value. The second variable describes the state the retriable future is in. `Retry` means the computation needs to be retried. `Stop` means we are done. `Idle` means that either a retry or stop can still take place.

We query a retriable future via `successR`. The `successR` operation is non-blocking because its main body is wrapped into a future. In the main body, we attempt to read a successful result. If available we will issue a stop of the retriable future. Both steps are executed atomically as part of an STM transaction which guarantees that we maintain a consistent view of retriable futures. If a successful result is not available, we issue a STM retry. See location `(L)`. The STM retry first leads to a roll back and then suspension of the STM transaction. We resume once the result of the retriable future changes.

```
data Res a     = Empty | Fail | Succ a
data State     = Idle | Retry | Stop
type RFuture a = (TVar (Res a), TVar State)

successR :: (a → IO ()) → RFuture a → IO ()
successR cb rf = do
    future $ do
          v ← atomically $ do
                   v ← readTVar (fst rf)
                   case v of
                     Succ x → do writeTVar (snd rf) Stop
                                 return x
                     _      → retry                          -- (L)
          cb v
          return $ Just ()

    return ()
```

Fig. 5: Retriable Futures - Definition and Success Query

Once the STM transaction commits, we return value `v` which will be passed to the callback function `cb`. To ensure that the transaction as well as the subsequent call `cb v` are non-blocking, we run both steps in a (standard) future. The associated computation of a future must be of type `IO (Just a)`. Hence, we find the somewhat redundant statement `return $ Just ()`.

Via `futureR` we can asynchronously execute a retriable computation. See Figure 6. We again employ standard futures to ensure that the associated computation is non-blocking and executed asynchronously. See the program location marked as `(E1)`.

Helper function `fromFutToRFut` maps any result that is recorded in f to the transactional variable representing the result of the retriable future. See the program locations marked as `(S1)` and `(F1)`. Function `fromFutToRFut` is non-blocking due to the use of the non-blocking future API operations `success` and `failure`.

Helper function `checkRetryState` checks for any pending retry. If the state is idle we simply suspend until the state changes. See location `(I1)`. In case the future shall be retried, we 'cancel' any existing result (by setting the result to `Empty`) and set the state to idle again. See location `(R1)`. Once the transaction commits, we execute some action. In case of a retry, we continue. In this context, `cont` is bound to local function `go` which means that repeat the earlier steps starting at location `(E1)`. Stop means stop, or `return ()` in Haskell terms.

3.2 Conditional Retry

Figure 7 contains the implementation of the conditional retry operator. We create a new retriable future which serves as a proxy. The proxy propagates any retry

```
futureR :: IO (Maybe a) → IO (RFuture a)
futureR comp = do
  sig ← newTVarIO Idle
  rf ← newTVarIO Empty
  let x = (rf,sig)
  let go = do f ← future comp                                    -- (E1)
              fromFutToRFut f x (return ()) go
  go
  return x

fromFutToRFut :: Future a → RFuture a → IO () → IO () → IO ()
fromFutToRFut f (x@(rf,_)) stop cont = do
      success
       (λv → do atomically $ writeTVar rf (Succ v)              -- (S1)
                checkRetryState x stop cont
              ) f
      failure
       (do atomically $ writeTVar rf Fail                       -- (F1)
           checkRetryState x stop cont
       ) f

checkRetryState :: RFuture a → IO b → IO b → IO b
checkRetryState (rf,sig) stop cont = do
  action ← atomically $ do
     x ← readTVar sig
     case x of
       Idle → retry                                             -- (I1)
       Retry → do writeTVar sig Idle                            -- (R1)
                  writeTVar rf Empty
                  return cont
       Stop → return stop
  action
```

Fig. 6: Retriable Futures - Non-Blocking Execution

or stop request from the outer to the inner future. If the result of the inner future satisfies the condition, the proxy adopts the result.

In detail, we create a helper future (see location `(H2)`) which checks upon the result of the inner future and the state of the proxy. If the proxy says stop we stop. See location `(S2)`. If the inner future fails, we issue a retry and reset the result. See location `(F2)`. The same happens if the resulting successful value does not satisfy the condition. See location `(NP2)`.

Otherwise, we propagate the result to the proxy. Next, we perform similar checks as carried out by `checkRetryState`. See the cases starting at location `(P2)`. If the proxy signals stop we we tell the inner future to stop. In case of a retry, we reset the results of the inner future and proxy and propagate the retry request to the inner future. In all other cases, we suspend until the state of the proxy changes.

```
retryR :: RFuture a → (a → Bool) → IO (RFuture a)
retryR (f,sig) p = do
  sig_new ← newTVarIO Idle
  f_new ← newTVarIO Empty
  let go = do
       action ← atomically $ do
          v ← readTVar f
          s ← readTVar sig_new
          case (v,s) of
            (_, Stop) → return $ atomically $ writeTVar sig Stop -- (S2)
            (Succ x, _) →
              if p x
               then return $ do
                       atomically $ writeTVar f_new (Succ x)
                       action ← atomically $ do
                          v_sig_new ← readTVar sig_new
                          case v_sig_new of                -- (P2)
                            Stop → do writeTVar sig Stop
                                      return $ return ()
                            Retry → do writeTVar f Empty
                                       writeTVar f_new Empty
                                       writeTVar sig Retry
                                       writeTVar sig_new Idle
                                       return go
                            _ → retry
                       action
               else return $ do                              -- (NP2)
                       atomically $ writeTVar f Empty
                       atomically $ writeTVar sig Retry
                       go
            (Fail, _) → do                                   -- (F2)
                       return $ do
                         atomically $ writeTVar f Empty
                         atomically $ writeTVar sig Retry
                         go
            _ → retry
       action
  future $ do go                                             -- (H2)
               return $ Just ()
  return (f_new,sig_new)
```

Fig. 7: Conditional Retry

3.3 Lifting of Standard Future Operations

A (standard) future operations is turned into a retriable future operation by performing some (lifting) transformations among the respective values. See Figure 8 where we consider the alternative operator.

```
altR :: RFuture a → RFuture a → IO (RFuture a)
altR ft1 ft2 =
  fromFutOpToRFut [ft1,ft2]
            (λ [ft1,ft2] → do f1 ← fromRFutToFut $ fst ft1
                              f2 ← fromRFutToFut $ fst ft2
                              f3 ← alt f1 f2
                              return f3)

fromRFutToFut :: TVar (Res a) → IO (Future a)
fromRFutToFut x = do
     p ← promise
     f ← future p
     future $ do
          action ← atomically $ do
                   v ← readTVar x
                   case v of
                     Empty → retry
                     Fail → return $ tryFail p              -- (F3)
                     (Succ y) → return $ do trySuccess p y  -- (S3)
                                            return ()
          action
          return $ Just ()
     return f

fromFutOpToRFut :: [RFuture a] → ([RFuture a] → IO (Future b)) →
IO (RFuture b)
fromFutOpToRFut fs futOperator = do
  sig ← newTVarIO Idle
  rf ← newTVarIO Empty
  let x = (rf,sig)
  let go = do f ← futOperator fs
              fromFutToRFut f x
                 (mapM_ ( λ(_,sig) →                        -- (D3)
                          atomically $ writeTVar sig Stop) fs)
                 (do mapM_ ( λ(_,sig) →                     -- (R3)
                          atomically $ writeTVar sig Retry) fs
                     go)
  go
  return x
```

Fig. 8: Lifting of Standard Operators

Helper function `fromRFutToFut` maps the value of a retriable future to a standard future. We make use of promises to set the (standard) future value at specific program points. See locations `(F3)` and `(S3)`. Helper `fromFutOpToRFut` is a generalization of the earlier function `fromFutToRFut` and deals future operators which may have operands. The difference is that we need to propagate the respective requests to operands. See locations `(D3)` and `(R3)`.

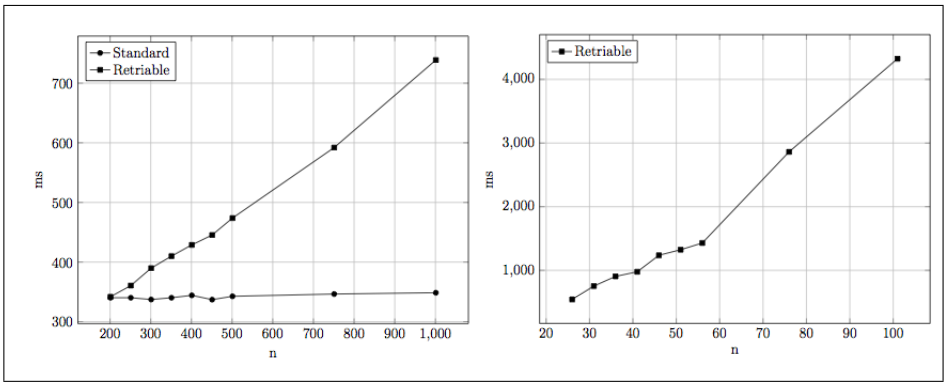

Fig. 9: Performance Measurements

The above lifting approach applies further future API operators such as any, combine etc.

3.4 Experiments

Each layer of retriable future introduces two additional transactional variables. To measure the overhead caused compared to standard futures, we consider some worst case examples. The first example is a deeply nested future expression of the following form.

$$x_1 = \mathsf{alt}(\mathsf{f_1}, (...(\mathsf{alt}(\mathsf{f_{n-1}}, \mathsf{f_n}))...))$$

We assume that futures f_1, ..., f_{n-1} yield failure instantly. Future f_n yields success after about 250ms. The second example includes a retry statement.

$$x_2 = \mathsf{retry}(\mathsf{alt}(\mathsf{f_1}, (...(\mathsf{alt}(\mathsf{f_{n-1}}, \mathsf{f_n}))...)))$$

We assume that *all* futures f_1, ..., f_n yield failure instantly. Future f_n yields success after about 250ms.

For each example, we measure the time until the successful result of either x_1 or x_2 becomes available. We have performed measurements for the Haskell as well as the Scala version. For the Haskell version, the measurements for x_1 show only small differences between our retriable future implementation and a standard implementation upon which retriable futures rely on.

In contrast, the performance of the Scala version seriously degrades for large n. See left part of Figure 9. The culprit seems to be the Scala implementation of STM. Profiling shows that a huge number of (JVM) threads are created, apparently, due to our use of the STM retry statement. The situation gets worse in case of the second example where all computations are effectively run in a busy loop. See right part of Figure 9. We believe that our implementation (in particular the Scala version) can be improved, e.g. by exploring alternatives to STM. This something we plan to pursue in future work.

4 Related Work and Conclusion

Futures are a well-studied concept [4, 3] with many implementations in various languages, e.g. Alice [11], Oz-Mozart [10], Scala [6] etc.

We argue to change the read-only semantics of futures to support retriable futures and gave several examples to support the need for such an extension. It appears that retriable futures are indeed a desirable feature as observed by several attempts to provide for 'retry patterns' in the Scala [7, 1, 13] and Java [8] setting. The focus of these works is on a single retriable computation whereas we consider more expressive operations as well.

To the best of our knowledge, we are the first to formally study retriable futures. We show how to obtain a fully working implementation of retriable futures by adding an STM-based retry management layer to an existing set of standard future operations. There is lots of scope for improvements as discussed in Section 2.4 and Section 3.4. This is something we plan to pursue in future work. We also intend to show correctness of our implementation by establishing a formal connection to the semantic description. In another direction, we plan to investigate the implications of a retry mechanism for related concepts such as join patterns [5], synchronous events [9] and reactive stream processing.

References

1. Pierre Andrews. Retry a fail-able block in scala. https://coderwall.com/p/5wetba, 2014.
2. Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, August 1977.
3. Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
4. Cormac Flanagan and Matthias Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, January 1999.
5. Cedric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proc. of POPL'96*, pages 372–385. ACM Press, 1996.
6. Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and promises. http://docs.scala-lang.org/overviews/core/futures.html.
7. Ayush Mishra. A simple way to implement retry support for akka future in scala. http://blog.knoldus.com/2013/08/18/a-simple-way-to-implement-retry-support-for-akka-future-in-scala/, 2013.
8. Tomasz Nurkiewicz. Asynchronous retry pattern. https://github.com/nurkiewicz/async-retry, 2014.
9. John H. Reppy. *Concurrent programming in ML*. Cambridge University Press, 2007.
10. Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.
11. The Alice Manual. http://www.ps.uni-sb.de/alice/manual/.
12. Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Proc. of LICS'92*, pages 162–173. IEEE, 1992.
13. Doug Tangren. because you should never give up, at least not on the first try. https://github.com/softprops/retry, 2013.