Prettier Concurrency

Purely Functional Concurrent Revisions

Daan Leijen Sebastian Burckhardt Manuel Fähndrich
Microsoft Research
{daan,sburckha,maf}@microsoft.com

Abstract

This article presents an extension to the work of Launchbury and Peyton-Jones on the *ST* monad. Using a novel model for concurrency, called *concurrent revisions* [3, 5], we show how we can use concurrency together with imperative mutable variables, while still being able to safely convert such computations (in the *Rev* monad) into pure values again.

In contrast to many other transaction models, like software transactional memory (STM), concurrent revisions never use rollback and always deterministically resolve conflicts. As a consequence, concurrent revisions integrate well with side-effecting I/O operations. Using deterministic conflict resolution, concurrent revisions can deal well with situations where there are many conflicts between different threads that modify a shared data structure. We demonstrate this by describing a concurrent game with conflicting concurrent tasks.

Categories and Subject Descriptors D.3.3 [*Concurrent Programming Structures*]; D.1.1 [*Functional Programming*]

General Terms Algorithms, Design, Languages

Keywords Concurrent Revisions, Isolation, Transaction

1. Introduction

Almost 16 years ago, Launchbury and Peyton-Jones describe in their seminal paper "State in Haskell" [23] how stateful computations in the *ST* monad can be safely converted into pure values when their side effects are unobservable. This is elegantly done by relying on parametricity and giving the *runST* combinator a rank-2 type. This article is in essence an extension of their result: using a novel model for concurrency, called *concurrent revisions* [3, 5], we can use concurrency together with imperative mutable variables, while still being able to safely convert such computations (in the *Rev* monad) into pure values again.

The basic idea of concurrent revisions is that on a *fork*, the entire shared state is (conceptually) copied such that each thread works in full isolation. Only at a *join* are the modifications that a thread made merged back. On a write-write conflict a merge function is called to resolve the conflicting writes, where the default strategy is to give priority to the writes of the joinee.

In contrast to many other transaction models, like software transactional memory (STM), concurrent revisions never use rollback and always deterministically resolve conflicts. As a consequence, concurrent revisions integrate well with side-effecting I/O operations. Also, due to the deterministic conflict resolution, concurrent revisions can deal well with situations where there may be many conflicts between different threads that modify a shared data structure. In Section 5.1 we demonstrate this by creating a concurrent game with conflicting concurrent tasks.

Specifically, we make the following contributions:

- Even though a revisional compution is fully concurrent and can mutate versioned variables, we can use the Haskell type system to turn such computation safely into a pure value again. The safety result is a direct consequence of the semantics of concurrent revisions which is deterministic no matter how the threads are scheduled [5] (Section 2.2)
- The type system also guarantees that versioned variables do not escape the scope of a revisional computation. This is done using the same technique as in the *ST* monad using a rank-2 polymorphic type (Section 2.2).
- Haskell makes it easy to implement custom versioned data types that can handle merge conflicts in special ways. We demonstrate the use of cumulative counter (Section 3), and the use of compensation tables to specify semantics for more complex datatypes (Section 3.2).
- We show how the conflict resolution of concurrent revisions can elegantly solve standard problems with transactional read- and write-skew [7, 2] (Section 4).
- In contrast to most transactional models, concurrent revisions never roll back, and therefore integrate well with I/O effects. The type system lets us freely use I/O operations within a revision, but such revisional computations can only be run in *IO* monad themselves (Section 5).
- We show the full implementation of a small game that executes all its basic operations concurrently, namely rendering, physics simulation, and player input. Due to the read-write and write-write conflicts between these operations traditional concurrency mechanisms like locks and STM do not deal well with such applications (Section 5.1).

We have a full implementation of concurrent revisions as a small library in Haskell (available at [14]) which we describe

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Haskell'11, September 22, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0860-1/11/09... $10.00

in detail in Section 6. All examples in this paper come with the library.

2. An overview of concurrent revisions

Concurrent revisions simplify the parallelization of conflicting tasks by conceptually copying the entire shared state on a fork. Tasks execute in complete isolation where each has its own copy of the shared state. This is somewhat analogous to a source control system that allows multiple programmers to work on code at the same time by creating local copies of the files. For this reason, we usually call tasks in our model *revisions*.

In order to guarantee safety through the type system, we execute all revisonal code in a revision monad *Rev*:

```
data Rev s α

instance Functor (Rev s)
instance Monad (Rev s)
```

A value of type *Rev s* α signifies a revisional computation with a result value of type α in a revisional heap *s*. As we will see, the heap parameter *s* is used just like in the *ST* monad to guarantee isolation of different revisional computations.

The shared state for a revision is constructed through *versioned variables*:

```
data Versioned s α

vcreate :: α → Rev s (Versioned s α)
vread :: Versioned s α → Rev s α
(=:) :: Versioned s α → α → Rev s ()

vmodify :: Versioned s α → (α → α) → Rev s ()
vmodify v f
  = do x ← vread v
       v =: f x
```

A value of type *Versioned s* α is a shared variable in heap *s* that stores a value of type α. The function *vcreate* creates a fresh versioned variable, *vread* reads the value stored, and (=:) assigns a new value. Revisions can be created using *fork*, and joined again with *join*:

```
data RevisionTask s α

fork :: Rev s α → Rev s (RevisionTask s α)
join :: RevisionTask s α → Rev s α
```

A call to *fork* returns a handle to the forked revision as a *RevisionTask s* α. The *join* function uses this handle to join on the specifed revision (which blocks until the revision is done). Using the above basic functions, we can write our first concurrent revision program in Haskell:

```
simple :: Rev s Int
simple =
  do vx ← vcreate 0
     vy ← vcreate 0
     r ← fork (vx =: 1)
     x ← vread vx
     vy =: x
     join r
     vread vy
```

In the above code, we first allocate two versioned variables *vx* and *vy* using *vcreate*. Then, we fork a concurrent revision that just assigns 1 to *vx* (eg. *vx* =: 1). In the main revision though we assign the value of *vx* to *vy*. After joining the child revision we return with the current value of *vy*.

What will the value of *vy* be at this point? In traditional forms of concurrency with shared variables, regardless of locks or atomic blocks, the value of *vy* is undetermined and could be either 1 or 0, depending on a particular scheduling of the threads. Not so with concurrent revisions: since each revision gets its own local copy of the state, the read *vread vx* always returns 0, and therefore the final value of *vy* is always 0 (and never 1, no matter how the revisions were scheduled). The concurrent assignment in the child revision, *vx* =: 1 is always fully isolated and within each concurrent revision, one can reason about inside each revision as if it was executed sequentially.

When a child revision is joined, the changes it has made are merged back into the main revision; this is as if the writes in the child revision all happened atomically at the join time. In our example, after the *join*, the assignment to *vx* in the child revision becomes visible in the main revision and the final value of *vx* is always 1.

For any write-write conflict, a special merge function is called that resolves the conflict. By default the merge function always lets the *writes of the joinee win*, but this behaviour can be customized on a per-type basis. For example, one might want to use cumulative integers where the additions in all revisions are added together. We discuss this in detail in Section 3. Note that in contrast to transactional memory for example, there is never a read-write conflict due to the full isolation of each revision.

2.1 Exceptions and abandonment

Exceptions can of course be raised in a revision. When one joins on an exceptional revision, the exception is re-raised by the *join* operation. However, none of the modifications done by the joinee are merged back into the main revision. A problem with exceptions in general is that an exceptional computation could leave the shared state in an inconsistent configuration where not all invariants hold. Since every revision works in complete isolation we can simply disregard any modifications of an exceptional revision and avoid the need to patch up incomplete state changes. For the same reason, we can also safely support operations to terminate a revision:

```
kill :: RevisionTask s α → Rev s ()
abandon :: RevisionTask s α → Rev s ()
```

Both operations forget about a revision but *kill* raises asynchronously an exception in the target revision, while *abandon* blocks until the target revision is done (and then disregards any modifications the target revision has done). Of course, one can also just never *join* on a revision but the above operations ensure that all revisions are properly garbage collected.

2.2 Safe encapsulation of revisions

As we saw in our previous example the final result of the computation was always the same. It turns out that, under some constraints, *any* concurrent revision program is actually fully deterministic despite its imperative variables and concurrent execution. In earlier work [5], we describe a precise operational calculus for concurrent revisions (based on the AME calculus by Ababi [1]). Using this calculus, we have proven various properties, including that the result of a revisional program is always deterministic.

Of course, this result only holds for revisional programs that have no I/O effect: if one can call the random number generator it is easy to construct a non-deterministic program. More insidiously, if a revisional program can take a lock, its result could depend on a particular scheduling.

Our Haskell implementation relies on the type system to guarantee that no arbitrary I/O takes place by restricting the operations that can execute in the *Rev*-monad. This ensures that all operations in the revision monad correspond directly to our calculus and we are guaranteed that such programs are determinate. As a consequence, we can safely convert such *Rev* computations into pure values:

```
revisioned :: (∀s. Rev s α) → α
```

Just like the *ST* monad, there is one catch: we need to ensure that we leak no versioned variables that could be shared among different revisional computations. We apply the same trick as in *runST* [23] by giving *revisioned* a rank-2 polymorphic type. By requiring that the revisional monad is polymorphic in its heap variable, we guarantee by parametricity that there is no versioned variable in the environment that can observe that heap. Take for example the following (ill-typed) program:

```
leftRight    – wrong
  = let v = revisioned (vcreate True)
    in revisioned (vread v) ||
       revisioned (do{ vmodify v not; vread v})
```

If this program were to be accepted by the type system, it could return either $False$ or $True$ depending on the evaluation order. Fortunately, the bottom two *revisioned* applications are rejected. Both expressions have type *Rev s Bool*, where the type variable s occurs in the type environment for the type of v :: *Versioned s Bool*. Therefore, both expressions are not polymorphic in s and cannot be passed as an argument to *revisioned* – correctly rejecting the program.

2.3 Sequential vs Concurrent reasoning

As we have seen, with concurrent revisions we can truly reason about each branch locally without considering any interleavings. However, at the same time there may not always be an equivalent sequential execution for a revisioned program! Consider the following example:

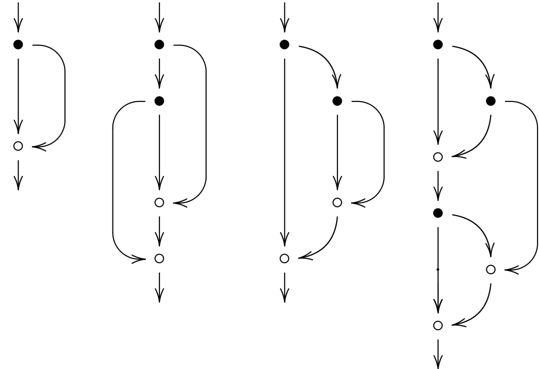
```
do vx ← vcreate 0
   vy ← vcreate 0
   r ← fork (do x ← vread vx
                when (x == 0) (vmodify vy (+1)))
   y ← vread vy
   when (y == 0) (vmodify vx (+1))
   join r
   x ← vread vx
   y ← vread vy
   return (x,y)
```

The example creates two versioned variables *vx* and *vy*. In the forked revision, we increment *vy* if *vx* is zero, and in the main revision we do the opposite, incrementing *vx* if *vy* is zero. Using revisions, the outcome is always determinate: both branches get their own local (conceptual) copy of the state, and both branches will increment the variables ending in (1,1). Now imagine that we execute the same program on a sequentially consistent machine using regular mutable variables. In that case there are three possible outcomes depending on the exact interleaving of the two threads, namely (0,1), (1,0), and (1,1). Interestingly, the (1,1) result is somewhat unexpected here since it requires a thread switch just after the child thread has read the x value (or the other way around). When using software transactional memory [9], the situation can be improved by using *TVar*'s and wrapping both concurrent actions inside an *atomically* statement. Still, the computation is non-deterministic and returns either (0,1) or (1,0).

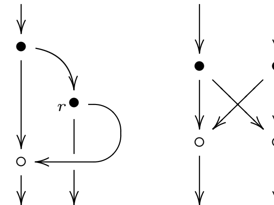
As the example shows, with concurrent revisions the answer is always determinate, but at the same time there is no sequential execution that gives the same result. The lack of equivalence to some sequential execution is no accident: requiring such equivalence fundamentally limits the concurrency that can be practically exploited if tasks exhibit conflicts. For the kind of applications we have in mind, conflicts may be quite frequent.

2.4 Revision diagrams

Comfortably reasoning about application behavior in the absence of serializability requires understanding and conceptualizing a nonlinear history of state. We achieve this by introducing revision diagrams that directly visualize how the shared state can be forked, updated, and joined. In previous work [5] we show that the revision diagrams have a formal and well-defined meaning with a direct correspondence to execution steps in the operational semantics. Here are some examples of possible revision diagrams:



In a diagram, we use a bullet • for fork nodes and a circle ∘ for join nodes. Revisions correspond to the vertical chains between a fork and join node in the diagram. These diagrams visualize clearly how information may flow (it follows the edges) and how effects become visible upon the join. As we can see, the structure of revision diagrams is more general than series-parallel graphs (SP graphs) since revisions can escape the lexical scope (as shown in the rightmost diagram). At the same time, revision diagrams are not arbitrary DAG's, and the following diagrams are not possible:



The left diagram is not possible since the main branch cannot join on the outer revision as the (fresh) outer revision handle r cannot be part of its (isolated) state. The rightmost diagram cannot be constructed for similar reasons, in

particular, we can prove that all valid revision diagrams are semi-lattices [5]. As we see later in Section 3 on merging, the semi-lattice property is very important since it ensures that there is always a least common ancestor value for any merge (which happens to always be the fork node of the joinee), corresponding to a three-way merge in version control systems.

Unfortunately, in the C# implementation of concurrent revisions [3] we have to resort to dynamic checks to confirm that there are no improper joins that would lead to an invalid revision diagram: this can be done for example by communicating the revision handle through a (non-versioned) shared memory location. In a side-effect free language like Haskell this is not possible: in a pure revision, the type system ensures statically that the revision handles of newly forked revisions must flow downward along the revision diagram and none of the invalid revision diagrams can be constructed.

Note that the structure of revision diagrams is entirely dynamic, not lexical. In particular, once a revision is forked, its handle can be stored in arbitrary data structures and be joined at an arbitrary later point of time. In some sense, revisions behave like futures whose side effects are delayed, and take effect atomically at the moment when the future is forced.

There is one more kind of invalid diagram: we can never join more than once on a revision. In earlier work [5] we show that this is essential for determinism or otherwise we could create revisional computations that depend on a particular scheduling. Any double join should always result in terminating the revisional computation. Fortunately, since revisions are deterministic, if any revision is joined more than once, this will always be the case. In our implementation we simply raise an exception whenever a revision is joined more than once.

2.5 Limitations

Because concurrent revisions are deterministic with respect to a particular schedule, we cannot easily describe computations that *rely* on arbitration: like a producer-consumer program or the Santa Claus problem [20]. Similarly, some concurrent algorithms rely on *not* being isolated, for example, visiting a graph concurrently using a shared *visited* flag in each node.

We believe that many algorithms, like graph visiting, can be described nicely using concurrent revisions too but need to be reformulated; much like phrasing an imperative algorithms in a functional setting. With regard to producer-consumer problems, we are currently working on extending the revision model to support such applications in the form of a distributed client-server model. An initial version can be found as part of the library [14].

3. Custom merging

When there is a write-write conflict at the time of a *join*, the default conflict resolution is to let the write of the joinee win. Perhaps somewhat surprisingly, this strategy works quite well in practice since it allows one to prioritize different revisions and resolve conflicts by choosing a specific join order. Sometimes though we need a more sophisticated merging strategy, for example when aggregating results in parallel.

3.1 A counter

A common versioned type is that of a *cumulative integer* or *counter*. The idea is that a counter has only one operation, namely *inc* which adds to the counter. When two revisions concurrently add to the counter, the join should merge all additions. This is a typical example of an aggregation variable. There are many examples of such types, for example cumulative sets merge using a union operation, or a high score merges using the *max* operation.

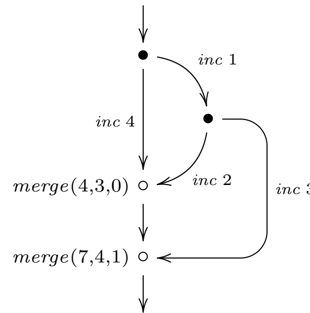
The *vcreateM* function allows one to specify a custom merge function. Using this function, we can specify a counter as an abstract data type with a *get* and *inc* operation:

```
data Counter s = Counter (Versioned s Int)

createCounter :: Int → Rev s (Counter s)
createCounter i =
  do v ← vcreateM merge i
     return (Counter v)
  where
    merge main joinee orig
      = main + (joinee − orig)

inc (Counter v) i = vmodify v (+i)
get (Counter v) = vread v
```

The merge function of the counter takes three arguments: the value in the *main* revision, the value of the *joinee*, and the original value *orig* from their least common ancestor point (which is always the value at the start of the joined revision). The merge function resolves the conflict by adding the delta *joinee* − *orig* to the value of the main revision. Here is a small example of a counter in action:

```
testCounter
  = revisioned $
    do c ← createCounter 0
       r1 ← fork (do inc c 1
                     r2 ← fork (inc c 3)
                     inc c 2
                     return r2)
       inc c 4
       r2 ← join r1
       join r2
       get c
```

The final result of *testCounter* is 10, i.e. the sum of all increment operations. This becomes more clear if we look at the annotated revision diagram for *testCounter*:



We have annoted the join nodes with the results of the merge operations. In particular, the second merge operation gets called with the value 1 from the least common ancestor node which allows the merge function to calculate the proper delta of the revision r_2.

We conclude by remarking that we can easily define *vcreate* in terms of *vcreateM* by explicitly specifying the default merge strategy:

```
vcreate x = vcreateM merge x
        where
          merge main joinee orig = joinee
```

Similarly, we can specify versioned variables where the main revision always wins:

```
vcreateMain x = vcreateM merge x
            where
              merge main joinee orig = main
```

3.2 Compensating actions

In practice, not all data types have such easy merge functions as counters, high scores, and cumulative sets. Some data types in particular combine both absolute operations (like *set*) with relative operations (like *add*). A prototypical example of such type is an *integer register* which we can set to a value, or add another value to. Specifying a semantics for such type in a concurrent setting is not entirely straightforward; for example, what should happen on the join if one revision sets the register while another adds to it concurrently?

One way of specifying semantics for such types is through so-called *compensation functions* [4]. A compensation function comp specifies for every pair of possible operations (m_1, m_2) what their compensating actions (n_1, n_2) are, such that composing $m_1; n_1$ is equivalent to composing $m_2; n_2$. This is exactly what we need for implementing a merge function: if a main revision did operation m_1, and the joinee m_2, we can specify the result of the merge as n_1. For example, for our *Counter* data type, the compensating function can be specified as:

$$\mathsf{comp}(add(i), add(j)) = (add(j), add(i))$$

Writing ϵ for an empty operation, and *assign* for assignment, we can write the default merge strategy for versioned variables where the joinee wins as:

$$\mathsf{comp}(assign(i), assign(j)) = (assign(j), \epsilon)$$

It turns out that from these one-operation compensation tables, we can always generalize them to compensation functions that work over arbitrary sequences of operations using tiling techniques [4].

For our register example, we need to specify how we can combine both *set* and *add* operations. For two *set* or *add* operations, we simply use the previous semantics:

$$\mathsf{comp}(set(i), set(j)) = (set(j),\ \epsilon)$$
$$\mathsf{comp}(add(i), add(j)) = (add(j),\ add(i))$$

For a *set* and *add*, we are going add bias:

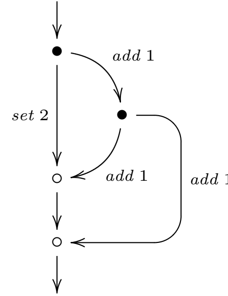
$$\mathsf{comp}(set(i), add(j)) = (add(j),\ set(i{+}j))$$
$$\mathsf{comp}(add(i), set(j)) = (set(j),\ \epsilon)$$

The philosophy behind the above semantics is that we want to keep the strategy where writes in the joinee win. When a joinee does a *set* operation, this overwrites anything the main revision did. But if the joinee just did additions, then all those additions are added to the value in the main revision. Note that other choices are possible: compensation tables are only a way to concisely describe a particular concurrent semantics which lends itself to validation. This becomes especially important with more complex data types like mutable lists for example [4].

Here is a small example that demonstrates a register:

```
testRegister = revisioned $
   do r ← createRegister
      r1 ← fork (do add r 1
                    r2 ← fork (add r 1)
                    add r 1
                    return r2)
      set r 2
      r2 ← join r1
      join r2
      get r
```

In a moment, we will see how we can implement the register, but for now we assume we have *createRegister* with the associated *add*, *set*, and *get* operations.What is the final value of this expression? Drawing a revision diagram helps us to gain more insight in the performed operations:

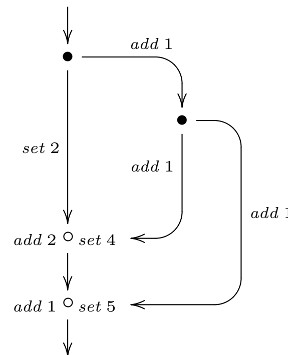


Since every forked revision just does relative operations (*add*), all those additions are added to the *set* value in the main revision, resulting in a final value of 5.

3.3 Eventual consistency

In the implementation, a merge function always uses the left-side of the compensating pair since we always merge into the main revision. So why specify the right component too? This allows us to annotate a revision diagram on the join points with both compensating actions. Now, we can follow *any* path through the diagram composing all operations we encounter along that path – choosing the left compensations along main paths, and right compensations along joinee paths. It turns out that, given a valid compensation function, that at a given end-point, the composition of all operations along any path to that point results in the same end state. This is in essence an 'eventual consistency' result for concurrent revisions. Moreover, it gives us back a form of sequential reasoning over concurrent diagrams: at any join point, we can disregard the concurrent joinee, and instead pretend that we atomically performed the compensating action instead.

Let's illustrate the above result with the register example:



Here we annotated our previous revision diagram at the join nodes with the compensating actions for each branch. There are three possible paths through the diagram, that result in the following sequences of operations:

- *set* 2; *add* 2; *add* 1
- *add* 1; *add* 1; *set* 4; *add* 1
- *add* 1; *add* 1; *set* 5

And as expected, any path results in the same final value.

3.4 Implementing the register

In principle, we could implement the custom merge function of the register using our compensation table, and then use the general tiling algorithm to apply the basic compensation table on arbitrary sequences of operations. However, that implies we would need to represent the register as a sequence of operations – it is much more efficient and natural to implement the register as a mutable variable.

To do so, we are going to represent the integer register with the following abstract data type:

```
data Register s = Register (Versioned s Reg)

data Reg = Rel Int Int | Abs Int

get (Register v) i
  = do reg ← vread v
       case reg of
         Rel base x → return (base+x)
         Abs x → return x

add (Register v) i
  = vmodify v (\reg → case reg of
                        Rel base x → Rel base (x+i)
                        Abs x → Abs (x+i))

set (Register v) i = vmodify v (\reg → Abs i)
```

The register can be either absolute, *Abs*, which means it was *set* at a certain point, or it can be relative *Rel* which means that only additions took place. We can see this clearly in the code for *set* which always returns an *Abs* value, regardless if it was relative or not. The *Rel* constructor has two integer fields, the first one is the base value, while the second one represents all additions that were done. As we can see in *get*, the value of such register is the addition of the base value to its additions, *base*+*x*.

The merge function can now be specified according to our semantics:

```
merge main joinee orig
  = case joinee of
      Abs x → Abs x
      Rel _ x → case main of
                  Rel base y → Rel base (x+y)
                  Abs y → Abs (x+y)
```

If the joinee used a *set* operation and is absolute, it disregards anything that the main revision did. If its operations were all relative though, the additions of the joinee are added to the value of the main revision. There is one more thing to take care of: in a freshly forked revision, we would like to always start afresh with a relative register, otherwise it would always seem as if we *set* the register. The *vcreateMF* function allows us to specify an extra operation that is called when a versioned variable is first accessed after a fresh fork:

```
createRegister :: Rev s (Register s)
createRegister
  = do v ← vcreateMF merge myfork
       return (Register v)
    where
      myfork reg
        = case reg of
            Abs x → return (Rel x 0)
            Rel base x → return (Rel (base+x) 0)
```

4. Read- and write-skew

The concurrent revision model is very similar to other transactional systems. In particular, we have shown in earlier work that concurrent revisions are a generalization of *snapshot isolation* [5]. Snapshot isolation is a guarantee that all reads made in a transaction will see a consistent snapshot of the state, and the transaction itself commits only if no updates it has made conflict with any concurrent updates made since that snapshot. Snapshot isolation has been adopted by many database systems, such as Oracle, PostgreSQL and Microsoft SQL Server. The adoption of snapshot isolation in these systems is mainly because it allows more scalable implementations than fully serializable transactions, yet it avoids most of the concurrency anomalies. There are two anomalies that still plague snapshot isolation, called read-skew and write-skew [7, 2]. In the next sections we will show that concurrent revisions do not suffer from this.

4.1 Write skew

We talk about write-skew when two transactions safely update separate variables x and y, but together violate an invariant that holds between x and y. As an example of write-skew we consider a bank account consisting of both a checking and savings account. The bank will charge a 1 dollar overdraft fee whenever the total balance of the accounts goes below zero. It is fine if only one of the accounts drops below zero, as long as the total is still positive. Below is a short example program (given by Fekete *et al.*[7]) that demonstrates write-skew:

```
testWriteSkew :: (Int,Int)
testWriteSkew
  = revisioned $
    do acc ← createAccount 70 80
       r1 ← fork (withdrawSaving acc 100)
       r2 ← fork (withdrawChecking acc 100)
       join r1
       join r2
       balance acc
```

The call *createAccount savings checking* creates a new bank account with the specified amounts for the savings and checking.[1] The customer then initiates two seperate transactions that both withdraw 100 dollars. Note that each transaction by itself should not cause an overdraft fee, but taken together the total balance becomes negative. In a system based on snapshot isolation the system will not charge an overdraft fee since each transaction sees a positive total balance [7]. Not so with concurrent revisions where the final account balance returned is $(-31,-20)$.

For concurrent revisions, we can use a custom merge function to charge an overdraft fee when the total balance becomes negative. To do so, it is important to version the

[1] How nice would it be to have such an API available!

savings and checking account together: since there is a condition that spans both accounts, they must be versioned as one such that concurrent changes to the separate accounts are properly detected as a conflict (and the merge function gets called). We can implement the abstract *Account* data type as:

```
data Account s = Account (Versioned s (Int,Int))

withdrawSaving (Account v) i
  = vmodify v (\(checking,saving) → (checking,saving−i))

withdrawChecking (Account v) i
  = vmodify v (\(checking,saving) → (checking−i,saving))

balance (Account v) = vread v

createAccount checking saving
  = do v ← vcreateM merge (checking,saving)
       return (Account v)
  where
    merge (main1,main2) (joinee1,joinee2) (orig1,orig2)
      = let new1 = main1 + (joinee1 − orig1)
            new2 = main2 + (joinee2 − orig2)
        in if (new1 + new2 < 0 && main1 + main2 ⩾ 0)
             then (new1−1,new2)
             else (new1,new2)
```

Since all deposits and withdrawals are cumulative, we use the same merge function for each account as for the *Counter* example. In the *merge* function, we first calculate the new balances using this merge strategy. After that calculation, we can easily check if the balance in the main revision became negative because of the join, and in that case we charge an overdraft fee.

4.2 Read skew

Read-skew is slightly more insidious that the write-skew. Again, we take an example from Fekete *et al.*[7] to illustrate the concept:

```
testReadSkew :: ((Int,Int),(Int,Int))
testReadSkew
  = revisioned $
      do acc ← createAccount 0 0
         r1 ← fork (depositSaving acc 20)
         r2 ← fork (withdrawChecking acc 10)
         join r1
         r3 ← fork (balance acc)
         bal1 ← join r3
         join r2
         bal2 ← balance
         return (bal1,bal2)
```

The program creates an empty account, and then initiates two concurrent transactions: one deposits 20 dollar on the saving account, while the other withdraws 10 dollar from the checking account. As we are unsure whether an overdraft fee is charged, we fork a third transaction to check the balance on the account bal_1 which will be (0,20). Finally, we join on the withdrawal transaction.

In a system based on snapshot isolation, the withdrawal transaction sees a balance of (0,0) and will therefore charge an overdraft fee (since the concurrent deposit is not seen yet). Unfortunately though, the customer can show a receipt of the intermediate positive balance of (0,20) and cause embarrassment. This form of read-skew is addresssed again through the merge function in concurrent revisions. Using our above implementation, the overdraft fee is only charged when merging into the main revision and since the balance never becomes negative, no overdraft fee is charged and the final result is ((0,20),(−10,20)) as expected.

5. Beyond purity: I/O and revisions

Untill now, we have restricted ourselves to purely functional concurrent revisions. This had the obvious advantage of being able to safely convert revisional computations into pure values. However, in contrast to many other concurrency models, concurrent revisions integrate well with I/O operations. Now, obviously, we cannot just add a *liftIO* operation:

```
liftIO :: IO α → Rev s α   -- wrong!
```

since that would allow us to run I/O operations inside a supposedly pure revisional computation, i.e.

```
unsafePerformIO io = revisioned (liftIO io)   -- wrong!
```

What we need to do is add an extra type parameter to the revisional monad that captures whether the revision is *Pure* or did *IO*:

```
data Pure :: ∗ → ∗   -- abstract
data Rev m s α

revisioned :: (∀s. Rev Pure s α) → α
revisionedIO :: (∀s. Rev IO s α) → IO α

liftIO :: IO α → Rev IO s α
liftRev :: Rev m s α → Rev IO s α
```

Other operations keep the type signatures they had and just become polymorphic in their monadic parameter, for example:

```
fork :: Rev m s α → Rev m s (RevisionTask s α)
join :: RevisionTask s α → Rev m s α

vcreate :: α → Rev m s (Versioned s α)
vread :: Versioned s α → Rev m s α
(=:) :: Versioned s α → α → Rev m s ()
```

Note how the type of *fork* enforces that the forking revision has the same monadic parameter as the forked revision: indeed, forking off a revision that performs *IO* operations should bring yourself into the *Ref IO s* monad too. In contrast, the *join* operation does not have such a constraint.

Using the *liftIO* function, we can now perform arbitrary I/O operations inside revisions (but we cannot convert such revisional compuations into pure values again). Usually, such extension wreaks havoc for many transactional concurrency models. For example, software transactional memory relies on *rollback* which cannot be supported for general I/O operations.

Concurrent revisions play very well with I/O operations: since no revision ever rolls back, I/O operations behave just as we would expect when running multiple threads. Of course, when using I/O operations, we lose the guarantee of determinancy but all versioned variables and merging behaves just like it did before. In particular, if concurrent I/O operations are isolated from each other, ie. do not take global locks, or use global variables, then a revisional computation is still determinstic with respect to any particular scheduling. If the I/O operations happen to be deterministic themselves, the outcome of a revisional computation is also deterministic.

```haskell
module Asteroids where
import Random
import Revision

-- Types ----------------------------------------
type Pos = (Int,Int)
type Player s = Versioned s Pos
type Asteroid s = Versioned s Pos

-- Constants ------------------------------------
maxX = 5
maxY = 4
coords = [[(x,y) | x ← [0..maxX]] | y ← [0..maxY]]

initialPlayer = (2,2)
initialAsteroids = [(0,0),(1,1),(2,3),(4,4)]

-- Main -----------------------------------------
play :: IO ()
play
  = revisionedIO $
      do player ← vcreate initialPlayer
         asteroids ← mapM vcreate initialAsteroids
         turn asteroids player

turn :: [Asteroid s] → Player s → Rev IO s ()
turn asteroids player
  = do r1 ← fork (playerTurn player)
       r2 ← fork (asteroidsTurn asteroids)
       r3 ← fork (wormhole asteroids)
       render asteroids player
       join r1
       join r2
       join r3
       checkWinner asteroids player

checkWinner asteroids player
  = do ps ← mapM vread asteroids
       p ← vread player
       if (p `elem` ps)
        then liftIO (putStrLn "You hit an asteroid!")
       else if (p == (0,0))
        then liftIO (putStrLn "You won!")
        else turn asteroids player

-- Render ---------------------------------------
render :: [Asteroid s] → Player s → Rev IO s ()
render asteroids player
  = do ps ← mapM vread asteroids
       p ← vread player
       let lines = map (map display) coords
           display pos = if p == pos then 'P'
                         else if pos `elem` ps then '*'
                         else '.'
       liftIO (mapM_ putStrLn lines)

-- Wormhole -------------------------------------
wormhole :: [Asteroid s] → Rev IO s ()
wormhole asteroids
  = do transport ← pick [False,True]
       when transport $
         do asteroid ← pick asteroids
            pos ← pick (concat coords)
            asteroid =: pos

-- Asteroid -------------------------------------
asteroidsTurn :: [Asteroid s] → Rev IO s ()
asteroidsTurn asteroids
  = mapM_ asteroidTurn asteroids

asteroidTurn asteroid
  = do moveit ← pick [False,True]
       when moveit $
         do direction ← pick [left,right,up,down]
            move asteroid direction

-- Player ---------------------------------------
playerTurn :: Player s → Rev IO s ()
playerTurn player
  = do c ← liftIO getChar
       case c of
         'l' → move player left
         'r' → move player right
         'u' → move player up
         'd' → move player down
         _ → return ()

-- Moving ---------------------------------------
left (x,y) = (x−1,y)
right (x,y) = (x+1,y)
up (x,y) = (x,y−1)
down (x,y) = (x,y+1)

-- update a position variable within the bounds
move :: Versioned s Pos → (Pos → Pos) → Rev m s ()
move vpos adjust
  = do old ← vread vpos
       let (x,y) = adjust old
       if (x < 0 || y < 0 || x > maxX || y > maxY)
        then return ()
        else vwrite vpos (x,y)

-- pick a random element from a list
pick :: [a] → Rev IO s a
pick sample
  = liftIO $
      do i ← getStdRandom (randomR (1,length sample))
         return (sample !! (i−1))
```

Figure 1. The Asteroids game uses concurrent revisions to concurrently execute the player input routine, the rendering, the asteroid simulation, and the wormhole simulation.

5.1 The asteroids game

To demonstrate the integration with I/O operations, we are going to implement a small game. In this game, the player has to navigate a field with asteroids. The full source code can be found in Figure 1. For demonstration purposes, we use simple ascii art to represent the player ship `P` and the asteroids `*`. A typical run of the program looks like:

```
TestAsteroids> play
*.....
.*....
..P...
..*...
....*.

u
*.....
.*P...
......
..*...
.....*

l
You hit an asteroid!
```

where the player moved first up (`u`) and then left (`l`). Also, the bottom asteroid moved one field to the right on its own. The game is turn based, and does four main operations per turn:

- The *playerTurn* asks the player to input a direction to move next.
- The *astroidsTurn* may randomly move an asteroid one field.
- The *render* function renders the current world.
- and finally, the *wormhole* function sometimes moves an asteroid from one place to a random other location.

The challenge is now that we would like to execute all four operations in parallel: ie. render the world, while asking the user for input, while updating the asteroids, while letting the wormhole transport asteroids through hyperspace. If an asteroid is both moved by the *asteroidsTurn* and the *wormhole*, the *wormhole* should get priority.

Note that this example is not far fetched: in most games one would like to render the world while doing physics simulation (e.g. moving the asteroids), while at the same time an opponent on the network may update state randomly (e.g. our *wormhole* process). The difficulties in particular are:

- The render phase and the *asteroidsTurn*/*playerTurn* function have a read-write conflict: one reads the position of all asteroids, while the other may update the position of the asteroids/player concurrently: we need to ensure that *render* sees a consistent snapshot. A pessimistic locking strategy (e.g. locks) does not help here since we would need to lock all the asteroids preventing any concurrency. Similarly, optimistic locking (e.g. STM) would neither work since the *render* function would always be rolled back due to conflicts, effectively serializing again. Another problem is that *render* performs observable I/O operations and rollback may not be possible.
- Similarly, the *asteroidsTurn* can have a write-write conflict with the *wormhole* if both happen to update the same asteroid. Here we could use fine-grained locking on each asteroid but it would be tricky to ensure that updates from the *wormhole* always get priority.

Concurrent revisions can deal elegantly with the above conflicts: there is never a read-write conflict since every revisions is isolated, and the write-write conflicts can be handled through merge functions. In our game, we represent the player and asteroids simply by versioned variables that hold a position:

```
type Pos = (Int,Int)
type Player s = Versioned s Pos
type Asteroid s = Versioned s Pos
```

As we can see in Figure 1, the main game loop is implemented as:

```
turn :: [Asteroid s] → Player s → Rev IO s ()
turn asteroids player
  = do r1 ← fork (playerTurn player)
       r2 ← fork (asteroidsTurn asteroids)
       r3 ← fork (wormhole asteroids)
       render asteroids player
       join r1
       join r2
       join r3
       checkWinner asteroids player
```

In particular, we fork a revision for every concurrent operation, and do the *render* task on the main revision (mainly because UI operations are usually tied to the main OS thread). Note how we prioritize *wormhole* updates over updates in the *asteroidsTurn* by joining the *wormhole* revision r_3 last. This works because we implemented the asteroid positions as regular versioned variables where the writes of the joinee win.

And this is it! In each of the concurrent operations, *playerTurn*, *asteroidsTurn*, *wormhole*, and *render*, we never have to consider any concurrent interactions (due to isolation), and we can reason about each as if they are executed sequentially.

5.2 Non-deterministic join

Since our game with the random asteroid movements is rather hazardous, it would be nice to add a continuous save game feature, where the game state is saved to disk. Of course, such operation may take quite some time so we need to run it concurrently. However, saving to disk may take so much time that we may not want to even join on it within a turn. What we need it is *tryjoin* operation that only joins on a revision once its done. Its type signature is:

```
tryjoin :: RevisionTask s α → Rev IO s (Maybe α)
```

The *tryjoin* function returns a *Maybe* type depending on whether it succeeded in joining or not. The monadic parameter of the revision monad must be *IO* now since *tryjoin* is non-deterministic and may depend on how the threads were scheduled.

To implement continuous saving of the game we pass in an extra revision handle $saver_0$ that correspond to the concurrent revision that saves the game. Whenever it is done, we immediately fork a new save revision again:

```
turn asteroids player saver0
  = do ...    -- fork/join as before
       done ← tryjoin saver0
       saver1 ← case done of
                  Nothing → return saver0
```

```
                    Just _ → fork (saveGame asteroids player)
        checkWinner asteroids player saver1
```

The *saveGame* function simply saves the game state to a temporary file:

```
import System.IO
import Control.Exception
...
saveGame :: [Asteroid s] → Player s → Rev IO s ()
saveGame asteroids player
  = do ps ← mapM vread asteroids
       p ← vread player
       liftIO $ do (_,h) ← openTempFile "." "save"
                   mapM_ (hPutStrLn h) (map show (p:ps))
                     `finally` hClose h
```

Just like before, the *saveGame* function can be reasoned about without considering any concurrent interactions, and it will always save games in a consistent state.

6. Implementation

We have implemented concurrent revisions as a Haskell library [14]. The implementation closely follows the algorithm described in earlier work[3]. Even though we have seen that concurrent revisions are deterministic, the internal algorithm relies heavily on destructive updates on mutable variables (*MVar*'s and *IORef*'s) and threads (*forkIO*). As such, we believe a more efficient implementation might be possible by supporting versioned variables directly in the runtime system. Since the basic algorithm is described in detail already [3], we concentrate in this paper mostly on the essential data structures and parts where the Haskell implementation differs.

Basic versioned variables are implemented as maps from version numbers to values. A versioned variable can also optionally store a set of custom operations *Ops*:

```
import qualified Data.Map as M

data Versioned s a = Versioned (MVar (VersionMap a))
                               (Maybe (Ops a))
type Version = Int
type VersionMap a = M.Map Version (IORef a)
```

Clearly, the implementation of the version map is rather inefficient by storing the entire map inside one *MVar*. More seriously, it prevents true concurrent access to such variables. A better implementation would be to store the version map as a mutable array of values indexed by versions:

```
data Versioned s a
      = Versioned (IOArray Int (Version,a)) ...
```

Since each revision will only access values of its version, all revisions can now concurrently access this array without any synchronization. There is one catch though: the size of the version map can become as large as the maximal number of concurrent writes to such versioned variables. Since this is not statically known, we may need to expand the array at runtime. Since this involves copying the array to a new (larger) array, we would have to synchronize all accesses again. The trick to circumvent this is to use nested arrays: one outer array which is only read by all revisions and which can be safely expanded, and inside it small fixed length arrays that contain the versioned values:

```
data Versioned s a
      = Versioned (IOArray Int FixedArray) ...
type FixedArray = IOArray Int (Version,a)
```

By choosing the size of the fixed length arrays to be 4, all elements in the nested array can be efficiently accessed by using modulo operations implemented using shifts. Unfortunately, the current implementation of *IOArray* induces a bit too much overhead in terms of space and time to make the nested array implementation worthwhile, even though it is more scalable. For the prototype library, we therefore opted to use the initial simple *MVar* implementation. If the versioned variables were implemented in the runtime system, we could of course use the more scalable nested array implementation.

In the implementation, the basic unit is not a revision, but a *segment*. These are basically the line segments of revision diagrams. What we call a revision is really a set of segments along a downward path. A revision therefore stores a reference to its root segment, and an updateable reference to its current segment (in which context it executes):

```
data Revision s = Revision{ revRoot :: Segment s
                          , revCur :: IORef (Segment s)
                          }
```

Note how the current segment is stored in an *IORef* which signifies that this reference is never accessed concurrently (or we would have used an *MVar* instead). Segments are defined as:

```
data Segment s
  = Segment{ segParent :: Segment s
           , segVersion :: Version
           , segRefCount :: MVar Int
           , segWritten :: IORef [VersionedAny s]
           }
```

Segments have a parent and a unique version. The *segRefCount* field holds the reference count which enables us to garbage collect revisions and versions of variables that are no longer needed. We use an *MVar* for the reference count since it can be accessed concurrently. Also, a segment maintains a list of versioned variables that were written in order to detect conflicts at join time. In order to have a homogeneous list of the versioned variables we need to use an existential type to hide the value type:

```
data VersionedAny s = ∀α. VersionedAny (Versioned s α)
```

We now have enough machinery to understand the implementation of the creation of versioned variables:

```
vcreate :: α → Rev m s α
vcreate x
  = unsafeLiftIO $
    do segment ← currentSegment
       ref ← newIORef x
       let vmap = M.singleton (segVersion segment) ref
       mvar ← newMVar vmap
       let versioned = Versioned mvar Nothing
       modifyIORef (segWritten segment)
                   (\ws → VersionedAny versioned : ws)
       return versioned
```

The creation function first gets the current segment and allocates a fresh version map with one entry indexed by the version of the current segment. It then adds itself to the write list of the current segment and returns. Two questions remain: where does the current segment come from and what about *unsafeLiftIO*?

The current segment is a thread-local variable. Every thread that we create will have an associated revision, which points to the current segment. To implement the thread local revision references, we unfortunately needed to resort to the use of *unsafePeformIO* in order to implement a global map from thread id's to revisions. Again, this may be improved if revisions would be part of the Haskell runtime system. The implementation of *currentSegment* now becomes:

```
revisions :: MVar (M.Map ThreadId (Revision s))
revisions = unsafePerformIO (newMVar M.empty)

currentRevision
  = do map ← readMVar revisions
       tid ← myThreadId
       case M.lookup tid map of
         Just revision → return revision
         Nothing → error "No current revision"

currentSegment
  = do rev ← currentRevision
       readIORef (revCur rev)
```

Of course, these functions should not be exposed directly to the user. Just like the *unsafeLiftIO* function which lifts any *IO* operation into a pure revision monad:

```
data Rev (m :: * → *) s a = Rev (IO a)

unsafeLiftIO :: IO a → Rev m s a
unsafeLiftIO io = Rev io

liftIO :: IO a → Rev IO s a
liftIO io = Rev io

liftRev :: Rev m s a → Rev IO s a
liftRev (Rev io) = Rev io
```

As we can see, the revision monad is just a wrapper around the *IO* monad. Since both s and m are phantom types [15], we need to annotate the m parameter with its kind $* \to *$. Executing the revision monad is now straightforward:

```
revisioned :: (∀s. Rev Pure s a) → a
revisioned action
  = unsafePerformIO (revisionedIO (liftRev action))

revisionedIO :: (∀s. Rev IO s a) → IO a
revisionedIO (Rev io)
  = do rev ← createRevision nullSegment
       bindRevision rev   -- sets the thread local revision
       io `finally` releaseRevision rev   -- .. and execute
```

It is somewhat unsatisfying that we need to resort again to the use of *unsafePerformIO* in the *revisioned* function. The reason is that even though we can prove that the exposed functionality of concurrent revisions is safe and deterministic, the internal implementation relies heavily on imperative updates and thread synchronization. This is not entirely unexpected though: for efficiency reasons, any pure functional programming language is usually itself implemented using imperative graph rewriting techniques.
The final data type to show is that of a revision handle:

```
data RevisionTask s α
  = RevisionTask (MVar (Revision s, Exc α))
                 (Chan (Revision s)) ThreadId

data Exc α = Ok α | Exc SomeException
```

A *RevisionTask* contains a *ThreadId* of the thread that executes the revision (to implement *kill*) and an *MVar* that holds both the revision and the final result of the computation, which is either the return value or an exception. The channel field (*Chan*) is used to implement distributed client-server revisions which we do not discuss in this paper. The *fork* operation creates a fresh revision, a fresh empty *MVar* and forks a new thread to execute the action. At the end of the action, the *fork* uses *putMVar* to put the created revision and final result into the *MVar*. The *join* operation simply blocks on this *MVar* to wait till the revision is done:

```
join :: RevisionTask s a → Rev m s a
join (RevisionTask done _ _)
  = unsafeLiftIO $
    do (joinee,exc) ← takeMVar done
       putMVar done (error "You can only join once")
       case exc of
         Exc e → throw e   -- rethrow exeptions
         Ok x → do merge joinee
                   return x
```

When the revisional computation is done and *takeMVar* returns, the modifications stored in the *joinee* revision are merged back in the current revision using *merge*. To handle the erronous situation where one tries to join more than once on a revision, we put back an error value into the *MVar*, such that any subsequent join on the same revision handle throws an exception.

7. Related work

Haskell and concurrency Haskell has a rich history of integrating concurrency: in a pure language, one can in principle execute any expression concurrently without changing the semantics. Parallel Haskell adds the *par* combinator where the programmer can specify when there are good opportunities for parallel execution [26, 16, 17, 27] More recent work on purely functional concurrency has concentrated on nested data parallelism with a focus on efficient multi-core execution [22, 6].

In contrast with purely functional parallelism, Concurrent Haskell supports 'regular' stateful concurrency with concurrent threads in the *IO* monad where data is shared through *MVar*'s and asynchronous channels [21] More recently, the work on software transactional memory [9, 20] showed how transactional memory computations (in the *STM* monad) could be safely combined with *IO* threads.

Transactions Just as we do with revisions, proponents of transactions have long recognized that providing strong guarantees such as serializability [19] or linearizability [11] can be overly conservative for some applications, and have proposed alternate guarantees such as multi-version concurrency control [18] or snapshot isolation (SI) [2, 7, 25]. In fact, revisions can be understood as a natural generalization of snapshot isolation, extended to handle resolution of write-write conflicts following some policy (as discussed in Section 4), and to support nesting. The relationship to snapshot isolation is discussed more formally in [5].

Isolation types Isolation types are similar to Cilk++ hyperobjects [8]: both use type declarations by the programmer to change the semantics of shared variables. Cilk++ hyperobjects may split, hold, and reduce values. Although these primitives can (if properly used) achieve an effect similar to revisions, they only provide determinacy guarantees

under certain restrictions. For instance, the following program may finish with x equal to either 2 or 1:

```
reducer_opadd<int> x = 0;
cilk_spawn { x++; }
if (x == 0) x++;
cilk_sync
```

because reading the hyperobject x (versus incrementing) violates one of the determinacy contraints. Also, Cilk++ tasks follow a more restricted concurrency model where the parallelism can be described as series-parallel directed acyclic graphs, or SP-dags [24].

Isolation types are also similar to the idea of transactional boosting, coarse-grained transactions, and semantic commutativity [10, 12, 13], which eliminate false conflicts by raising the abstraction level. Isolation types go farther though: for example, the type *Versioned* $s\ \alpha$ does not just avoid false conflicts, but resolves true conflicts deterministically (in a not necessarily serializable way).

8. Conclusion and Future work

We have shown how concurrent revisions and Haskell are a great match, where the type system can guarantee that concurrent programs that use mutable variables can be turned safely into pure values again (where possible). Moreover, the seamless integration with *IO* operations makes the model quite attractive to implement interactive and reactive programs that mutate shared state.

What concurrent revisions described in this paper cannot do well is perform computations that rely on arbitration: like a producer-consumer program or the Santa Claus problem [20]. We are currently working on extending the revision model to support such applications in the form of a distributed client-server model. An initial version can be found as part of the library [14].

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *Principles of Prog. Lang. (POPL)*, pages 63–74, 2008.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of SIGMOD*, pages 1–10, 1995.

[3] Sebastian Burckhardt, Alexandro Baldassion, and Daan Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA'10*, October 2010.

[4] Sebastian Burckhardt, Manuel Fähndrich, and Daan Leijen. Roll Forward, not Back – A Case for Deterministic Conflict Resolution. In *The 2nd Workshop on Determinism and Correctness in Parallel Programming*, 2011.

[5] Sebastian Burckhardt and Daan Leijen. Semantics of concurrent revisions. In *European Symposium on Programming (ESOP'11)*, 2011.

[6] Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. Nepal – Nested Data-Parallelism in Haskell. In *Euro-Par 2001: Parallel Processing, LNCS 2150*, pages 524–534, 2001.

[7] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[8] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Sym. on Par. Algorithms and Architectures (SPAA)*, pages 79–90, 2009.

[9] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, 2005.

[10] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, 2008.

[11] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[12] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *Principles of Programming Languages (POPL)*, pages 19–30, 2010.

[13] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Chew. Optimistic parallelism requires abstractions. In *(PLDI)*, 2007.

[14] Daan Leijen. Haskell revisions. Implementation available at `www.research.microsoft.com/~daan/hsrevisions`, 2011.

[15] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, October 1999. Also in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

[16] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. Seq no more: Better strategies for parallel haskell. In *3rd Haskell Symposium*, pages 91–102, September 2010.

[17] Simon Marlow, Simon L. Peyton-Jones, and Satnam Singh. Runtime support for multicore Haskell. In *ICFP 2009 — Intl. Conf. on Functional Programming*, pages 65–78, Edinburgh, Scotland, August 2009.

[18] P.A.Bernstein and N.Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[19] P.A.Bernstein, V.Hadzilacos, and N.Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[20] Simon Peyton Jones. Beautiful concurrency, 2007. Appears as a chapter in "Beautiful Code: Leading programmers explain how they think" (O'Reilly).

[21] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, January 1996.

[22] Simon Peyton Jones, Roman Leshchinskiy, Gabrielle Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, 2008.

[23] Simon L Peyton Jones and John Launchbury. State in Haskell. *Lisp and Symbolic Comp.*, 8(4):293–341, 1995.

[24] K. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, May 1998.

[25] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2006.

[26] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[27] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable implementation of Haskell. In *Programming Language Design and Implementation*, May 1996.