Acta Cybernetica **22** (2015) 151–169.

Derivable Partial Locking for Algebraic Data Types*

Boldizsár Németh† and Zoltán Kelemen†

Abstract

Parallelism and concurrency are one of the most actively researched fields in Computer Science. Writing concurrent programs is challenging because of the need for synchronization and solving possible race conditions and deadlocks while avoiding unnecessary waiting and overhead.

The integrity of the program data can be archived by providing locks for its data structures or using concurrent data structures. Partial locking allows threads to lock exactly those parts of the global data they need to read or update.

This article presents a method that helps the implementation of thread-safe programs with Algebraic Data Types [1]. By transforming the data model of the application to thread-safe data structures with a built-in, configurable locking mechanism including partial locking. With this support, the programmer can focus on the business logic of his application when writing the program. As part of this article, we prove that the shared version of the calculation will produce the same result as the original one.

Keywords: concurrency, partial locking, functional programming, Algebraic Data Type (ADT), representation synthesis, type transformation

1 Introduction

Programming parallel algorithms is complicated when the representation of business data is mixed with different synchronization primitives that must be used with their own separate handling functions.

We offer a solution to transform program data into a representation that is thread-safe, in the functional language Haskell. Our goals are to enable the programmer to parallelize a program without having to reimplement parts of the solution, and change how the representation can be used by multiple threads.

The solution is composed of generating the type of the shared representation, transforming values into the representation, generating accessors for the shared representation and functions that act like the constructors of the original data

*This work was supported by Ericsson Hungary and EITKIC 12-1-2012-0001.

†Eötvös Loránd University, Faculty of Informatics, Department of Programming Languages And Compilers. E-mail: {nboldi,kelemzol}@elte.hu

DOI: 10.14232/actacyb.22.1.2015.10

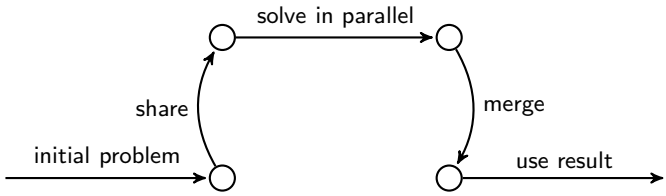

Figure 1: Sharing version of a program that calculates a single result

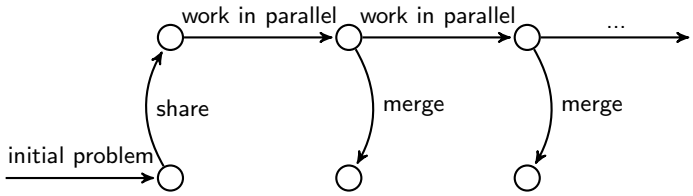

Figure 2: Sharing version of a program that runs continuously

type. Generating constructor functions and references helps the programmer to adapt his program for concurrent execution.

To be able to quickly change how multiple threads can work on the data structure. This paper describes a method to control the way program data is shared. This information is separated from the program logic. This is called the sharing configuration of the program. The configuration declares how the access to the program state is controlled.

Every type can be configured independently to define the protection for instances of the given type in the database (the state of the program). In our solution, the synchronization in shared programs is data-centric. The synchronization primitives are parts of the data structure. This way, the usage of these primitives can be guaranteed.

The execution of a program using our sharing method is done in three phases, as seen on Figure 1.

1. The problem is analyzed and the correct initial state of the program is shared.

2. The problem is solved by a number of threads working on the shared representation.

3. The result state is merged.

If the program runs continuously its state can be inspected any time while it is running, as seen on Figure 2.

1.1 Background

Hawkins et al. described a system [2] where data structures are synthesised from an abstract relational specification and a decomposition. A relational specification is similar to the pure ADT we use and decomposition is similar to our configuration. They applied the theory to concurrency in a subsequent paper [3].

We emphasize controlling the granularity at which the generated parallel computation locks objects. It can have a significant impact on the overall performance. This topic is thoroughly inspected at the context of automatically parallelized programs in [4].

Our method is practice-oriented, and that was inspired by Simon Marlow's book on the practical side of parallel programming in Haskell [5]. It has excellent chapters on MVars and Threads, and parallel programming using threads.

Other technologies exist for parallelizing algorithms. The Java streams [6] and C# PLINQ [7] can be used to parallelize a collection. Each thread is working on a separate part of the collection. Our goal is to parallelize heterogeneous data structures, not just homogeneous collections, and to enable different threads to access the same data in a thread-safe way.

Combining parallelism with partial locking as a method to enable multiple threads to access different parts of a data structure is a well known and thoroughly researched topic in the field of database design [8].

Data structures in Haskell can be defined by Algebraic Data Types. A simple ADT has the form `TCt` $\texttt{tv}_1$ ... $\texttt{tv}_n$ `=` `DCt` $\texttt{t}_1$ ... $\texttt{t}_n$. TCt is the type constructor, tv_i are the type variables, DCt is the data constructor and t_i are the types of the arguments of the constructor. The type variables can appear inside the types of the arguments. On the right side there can be more constructors, separated by `|`.

From this point, this paper will assume that the reader has some understanding of concurrency, threads, polymorphism and Algebraic Data Types.

2 Partial locking

Partial locking is a way to share a data structure and enable working threads to lock parts of it without interfering with each other. This can improve the scalability and performance of the whole program, because it reduces the time each thread spends waiting. The resulting data structure still ensures consistent use by multiple threads.

The shared database supports two kinds of locking: read locks and write locks. Reader threads can access the same part of the data concurrently. A thread cannot lock a part of the data for writing if another thread reads or writes the data or some part of it at that time. If the system is configured right, each thread will only lock the data part it needs for the computation.

For example, take a list of some type. If different elements of the list should be operated on independently but elements can only locked by one thread, it can be manipulated by partial locking.

2.1 Producing shared representation of a data type

Lets define the following data types for constructing the shared representation.

Definition 1. *Elements of the Skeleton representation*

- *Mem_n is a simple tuple type for collecting* n *members of arbitrary types. We defined a function for constructing them. (In the following definition the left side is the type declaration and the right side is the constructor definition. In this case the type name and the constructor name are the same.)*

  $\texttt{Mem}_n\ \texttt{t}_1\ \texttt{...}\ \texttt{t}_n\ \texttt{=}\ \texttt{Mem}_n\ \texttt{t}_1\ \texttt{...}\ \texttt{t}_n$
  $\texttt{mem}_n\ \texttt{=}\ \texttt{Mem}_n$

- *Alt_n is a union type that can have* n *states each having a value of possibly different types. The function* $\texttt{alt}_{n;i}$ *creates the* i*th alternative of the possible* n*.*

  $\texttt{Alt}_n\ \texttt{t}_1\ \texttt{...}\ \texttt{t}_n\ \texttt{=}\ \texttt{Alt}_{n;1}\ \texttt{t}_1\ \texttt{|}\ \texttt{...}\ \texttt{|}\ \texttt{Alt}_{n;n}\ \texttt{t}_n$
  $\texttt{alt}_{n;i}\ \texttt{=}\ \texttt{Alt}_{n;i}$

- *Skeleton is a composition of the Mem_n and Alt_n types.*

The Skeleton representation is a subset of ADT types.

2.2 Node types

There are five different node types, each representing a unique way to control accessing the shared representation between threads. As it can be seen on Figure 3, they can be ordered by how much paralellism they allow. Of course, more powerful locking mechanisms have a higher cost in terms of computation and memory overhead.

- *Clean* states that the configured data type must not be converted into a shared representation. *Clean* data cannot be accessed by multiple threads. *Clean* data in itself is immutable, but it can be part of a mutable database. *Clean* representation has no additional costs.

- *Noth* provides no extra protection for the data. It is similar to *Clean*, but the parts of the data configured to *Noth* can have a different configuration.

- *Prim* guarantees mutual exclusion for a part of the data. At any time of the execution, only one thread can have access to a part of the database that is configured to *Prim*. Threads trying to access the protected data will queue up, and access the resource in a first-come-first-served (FCFS) order. *Prim* is relatively low-cost, it is implemented by one synchronization primitive.

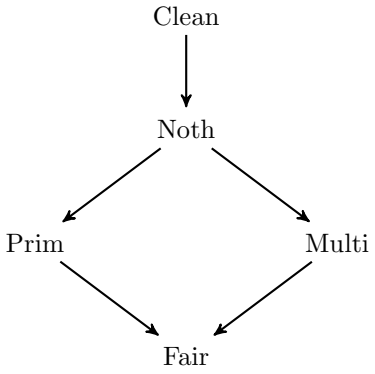

Figure 3: Different levels of support for multi-threading

- *Multi* allows multiple threads to access the data structure if their actions do not interfere. This applies to multiple reader threads or threads updating different parts of the data. However, it does not guarantee fairness. If a thread would like to gain exclusive access to the data, it may have to wait forever, if threads that can share the data access it frequently. *Multi* is more expensive than *Prim*.

- *Fair* provides a protection that allows non-interfering threads to work simultaneously and guarantees fair FCFS access for all threads. Fair nodes use concurrency primitives to implement a shared lock [9], but their cost is higher than that of *Multi* nodes.

It is clear that a good configuration strategy is needed to reach an optimal performance. Common sense dictates that nodes on the upper level of the representation are configured to high-level types and nodes on the lower level (that appear in higher numbers) are configured to types with low cost. Typically predefined types like `Char`, `String` and `Int` are configured to *Clean*.

As it was mentioned, each type can have different configurations. If different instances of the given type should be given different protection levels, the original type should be configured to the neutral node type `Noth` and wrapper data structures can be introduced with a different configuration. There is no default configuration, so each type that can be a part of the shared program state must be configured.

3 Context dependent references

References are first-class functional accessors that enable getting, setting and updating the accessed value in a context. They are implemented as a package on the package repository Hackage [10]. A reference represents a method to access information from a given object. They are interchangeable when their types are

the same. So a reference that accesses some information of type A (the accessed element) through an object of type B (the context) can be replaced by another reference for A through B.

Definition 2. *Reference*
Let `ref : s`$^{w}\triangleright$ `a`^{r} *be a reference iff*

- `s`, `a` *are types.* `s` *is the type of the context and* `a` *is the type of accessed element.*
- `w` *and* `r` *are type constructors and monads [11].* `w` *is called writer monad,* `r` *is called reader monad.*
- `get ref :: s → r a`
- `set ref :: a → s → w s`
- `update ref :: (a → w a) → s → w s`

Definition 3. *Composition of references*
Lets define the composition of two references marked with the `&` *operator:*
`& : a`$^{w}\triangleright$ `b`^{r} $\rightarrow$ `b`$^{w}\triangleright$ `c`^{r} $\rightarrow$ `a`$^{w}\triangleright$ `c`^{r}

- `get (r`$_1$ `& r`$_2$`) s` $\equiv$ `get r`$_1$ `(get r`$_2$ `s)`
- `set (r`$_1$ `& r`$_2$`) x s` $\equiv$ `update r`$_1$ `(set r`$_2$ `x) s`
- `update (r`$_1$ `& r`$_2$`) f s` $\equiv$ `update r`$_1$ `(update r`$_2$ `f) s`

Definition 4. *Member reference*
Let the reference `m : s`$^{w}\triangleright$ `a`^{r} *be the* j*th member reference iff*

- `s` *has only one constructor:* `s` $\equiv$ `Ctr t`$_1$ `... t`$_n$
- `get m (Ctr t`$_1$ `... t`$_j$ `... t`$_n$`)` $\equiv$ `return t`$_j$
- `set m b (Ctr t`$_1$ `... t`$_j$ `... t`$_n$`)`
  $\equiv$ `return (Ctr t`$_1$ `... b ... t`$_n$`)`
- `update m f (Ctr t`$_1$ `... t`$_j$ `... t`$_n$`)`
  $\equiv$ `return (Ctr t`$_1$ `... (f t`$_j$`) ... t`$_n$`)`

Member references can have arbitrary w and r parameters, but they cannot perform actions except for returning the referenced value.

Definition 5. *Structural reference A reference is a structural if it is a member reference or it can be created by a composition (by the* `&` *operator) of member references.*

A structural reference accesses some information that is inside the object.

4 Automatic generation of shared references and constructors

Because of the complexity of the shared representation, it would not be a feasible solution to burden the programmer with rewriting his code to use the shared representation. Instead we provided a way to generate constructor functions and references for the shared representation of the data structure, according to the constructors of the original data structure and normal references that can be generated for accessing its fields. The bodies of these references are generated according to the configuration of the types that are part of the representation.

For this reason defining a number of simple references for the node types, accessing the protected data.

The implementation of this automatic generation of references and constructors is using Template Haskell (TH) [12]. TH is a library for generating and inspecting the Abstract Syntax Tree (AST) of a Haskell program. It can only add new elements to the program, but cannot change the already defined parts. Program fragments can be inspected by looking them up using their names with the `reify` function, but the implementation of functions cannot be seen. To access the implementation, TH can process parts of the AST by receiving them directly as an argument.

We decided to store the configuration as instances of type families, because it can be queried from TH easily.

4.1 Generating references

The generated references are composed of two parts. The first part is a tuple reference for the index of the member accessed by the reference. Tuple references are simple means to access the nth field of a simple data structure parametrized by the types of it's members (for example, a pair, triplet, and so on).

The second part is a reference accessing the protected data from a node type. These references will be generated by inspecting the configuration for a given type, and can access the data in a protected, thread-safe way, depending on which kind of protection does the representation offer. These are the `_clean`, `_noth`, `_prim`, `_multi` and `_fair` references.

For example, given a normal representation of a log record with a configuration:

```
data Log
  = Log { _msg         :: String
        , _loggingDate :: Time
        }

type instance Node String = Clean
type instance Node Time = Prim
```

Calling the generator function `makeSharedRefs` with the quoted name of the `Log` datatype will create the following references:

```
msg :: Simple IOLens (Shared Log) String
```

```
msg = _1 & _clean
loggingDate :: Simple IOLens (Shared Log) (Shared Time)
loggingDate = _2 & _prim
```

4.2 Generating constructors

The generated constructors provide a way to build up larger shared data structures from smaller ones. The alternative method is to build up the data structure in its original representation and have to share it after that.

The shared constructor takes the shared arguments, creates the protection primitives for thread-safety and builds up the shared structure of the data to contain these protection primitives.

For example, lets inspect what kind of constructor functions would be generated from a list of log messages, if `[Log]` (list of Logs) is configured to `Multi` and `Log` is configured to `Prim`. First lets take a look on the original definition of the list: `data [a] = [] | a:[a]`. The type variable `a` will be replaced by `Log`. From `[]` the `%[]` operator will be generated, and from `(:)` the `(%:)` operator will be created. Finally the following functions will be generated after calling `makeSharedCons` on the quoted type expression of `[Log]`.

```
(%[]) :: IO (Shared [Log])
(%[]) = return (Alt2_1 Mem0)
(%:) :: Shared Log → Shared [Log] → IO (Shared [Log])
(%:) x xs = do d ← newPrim x
               ds ← newFair xs
               return (Alt2_2 (Mem2 d ds))
```

It was an implementation challenge to enable the user to create constructor functions for concrete types, because the original constructors belong to the general type. But it was solved by taking the types of the constructors and transforming them by replacing the type variables with their actual types.

5 Formal definition of transforming types and values to their shared representation

Lets define a transformation of a type in a general way, according to a given type mapping C. We will refer to this transformation with Trf_C.

Definition 6. *Generic transformations of types*

$\texttt{Trf}_C$: Raw $\rightarrow$ $\texttt{ExtendedSkeleton}_C$

Where Raw *is the set of all types, and* $\texttt{ExtendedSkeleton}_C \subseteq$ Raw *is the set of all types that can be constructed using* $\texttt{Alt}_n$, $\texttt{Mem}_n$ *applying the type-level function* C*. The function* C *has the type of* Raw $\rightarrow$ $\texttt{ExtendedSkeleton}_C$*.*

If T *is a scalar type,* $\texttt{Trf}_C$ T = T

If `T` *is not a scalar type, then* T *has* n *constructors with* r_1, r_2 ... r_n *fields:*
`T = Ctr`$_1$ `m`$_{1;1}$ ... `m`$_{1;r_1}$ | ... | `Ctr`$_n$ `m`$_{n;1}$ ... `m`$_{n;r_n}$
In this case

`Trf`$_C$ `T = Alt`^{n} (`Mem`$_{r_1}$ (`C m`$_{1;1}$) ... (`C m`$_{1;r_1}$))
... (`Mem`$_{r_n}$ (`C m`$_{n;1}$) ... (`C m`$_{n;r_n}$))

Definition 7. *Generic transformation of values*

We should also define the corresponding transformation of values, according to a given value-level function c:

`trf`$_c$ `::(t` $\rightarrow$ `Trf`$_C$ `t)` *where* c *is* `s` $\rightarrow$ `Trf`$_C$ `s`

If t *is a scalar type,* `trf`$_c$ `v = v`

Otherwise if t *has* n *different constructors,*

`trf`$_c$ (`Ctr`$_i$ `m`$_1^i$ ... `m`$_{r_i}^i$) = `alt`$_i^n$ (`mem`$_{r_i}$ (`c m`$_1^i$) ... (`c m`$_{r_i}^i$))

Definition 8. *Mappings for sharing*

And then we can define the mappings that give sharing as our transformation: Let `conf` *be a configuration given by the user, declaring how much multi-threading support is needed for the configured types.*

```
Sh_conf t = t                          if conf t = Clean
Sh_conf t = Trf_Sh_conf t              if conf t = Noth
Sh_conf t = Prim (Trf_Sh_conf t)       if conf t = Prim
Sh_conf t = Multi (Trf_Sh_conf t)      if conf t = Multi
Sh_conf t = Fair (Trf_Sh_conf t)       if conf t = Fair

sh_conf ::t -> Sh_conf t
sh_conf v = v                          if conf t = Clean
sh_conf v = trf_sh_conf v              if conf t = Noth
sh_conf v = newPrim (trf_sh_conf v)    if conf t = Prim
sh_conf v = newMulti (trf_sh_conf v)   if conf t = Multi
sh_conf v = newFair (trf_sh_conf v)    if conf t = Fair
```

Note: use of IO monad and its binding is omitted for the sake of simplicity.

```
Share = Trf_Sh_conf
share = trf_sh_conf
```

6 Proof of the semantic equivalence between the original and the shared version

Definition 9. *Homomorphism*

The function family h *is a homomorphism from* A *to* B *for references* r_1, r_2 ... r_n *iff*

- $h = (h_s, h_r)$, *where* $h_s : A \rightarrow B$ *is the mapping of original values and* $h_r : A \triangleright a \rightarrow B \triangleright a$ *is the mapping of references.*

- *For every reference* $r_i : A \triangleright a$ *where* $i \in [1, n]$
  - $\texttt{get r}_i \equiv \texttt{get (h}_r\ \texttt{r}_i\texttt{)} \circ \texttt{h}_s$
  - $\texttt{h}_s \circ \texttt{set r}_i\ \texttt{x} \equiv \texttt{set (h}_r\ \texttt{r}_i\texttt{) x} \circ \texttt{h}_s$
  - $\texttt{h}_s \circ \texttt{update r}_i\ \texttt{g} \equiv \texttt{update (h}_r\ \texttt{r}_i\texttt{) g} \circ \texttt{h}_s$

Theorem 1. *Homomorphism and composition*

If $(h_{s,1}, h_{r,1})$ *is a homomorphism for a set of references* R_1 *and* $(h_{s,2}, h_{r,2})$ *is a homomorphism for a set of references* R_2 *then* (h_s, h_r) *(*$h_s = h_{s,2} \circ h_{s,1}$*,* $h_r = h_{r,2} \circ h_{r,1}$*) is a homomorphism for references* $\{\texttt{r} : \texttt{r} \in R_1$ *and* $\texttt{h}_{r,1}\ \texttt{r} \in R_2\}$

Proof. The proofs for the equations of `get`, `set` and `update`:

```
get (h_r r) (h_s s)
≡ get ((h_r,2 ∘ h_r,1) r) ((h_s,2 ∘ h_s,1) s)
≡ get (h_r,2 (h_r,1 r)) (h_s,2 (h_s,1 s))
≡ get (h_r,2 r') (h_s,2 s')
≡ get r' s'
≡ get (h_r,1 r) (h_s,1 s)
≡ get r s
```

```
set (h_r r) x (h_s s)
≡ set ((h_r,2 ∘ h_r,1) r) x ((h_s,2 ∘ h_s,1) s)
≡ set (h_r,2 (h_r,1 r)) x (h_s,2 (h_s,1 s))
≡ set (h_r,2 r') x (h_s,2 s')
≡ h_s,2 (set r' x s')
≡ h_s,2 (set (h_r,1 r) x (h_s,1 s))
≡ h_s,2 (h_s,1 (set r x s))
≡ (h_s,2 ∘ h_s,1) (set r x s)
≡ h_s (set r x s)
```

```
update (h_r r) f (h_s s)
≡ update ((h_r,2 ∘ h_r,1) r) f ((h_s,2 ∘ h_s,1) s)
≡ update (h_r,2 (h_r,1 r)) f (h_s,2 (h_s,1 s))
≡ update (h_r,2 r') f (h_s,2 s')
≡ h_s,2 (update r' f s')
≡ h_s,2 (update (h_r,1 r) f (h_s,1 s))
≡ h_s,2 (h_s,1 (update r f s))
≡ (h_s,2 ∘ h_s,1) (update r f s)
≡ h_s (update r f s)
```

These three equations prove that the composition of homomorphisms is also a homomorphism. □

Definition 10. *Transformation of references (*`refShare`*)*

As already seen in Section 4.1, a structural reference $\texttt{ref : s}^{w} \triangleright \texttt{a}^{r}$ *can be transformed into a distributed representation.*

If the reference `ref` *is a structural reference composed of member references* m_1, ..., m_n *then* `refShare (m1 & ... & mn) ≡ refShare m1 & ... & refShare mn`

If ref *is a member reference accessing the* j*th field of the constructor* `refShare ref ≡ _j & protRef`$_{conf}$ `b`.

`protRef`$_{conf}$ *creates the protection primitive generated by* `share` *and is defined as follows:*

```
protRef_conf v = _clean   if conf t = Clean
protRef_conf v = _noth    if conf t = Noth
protRef_conf v = _prim    if conf t = Prim
protRef_conf v = _multi   if conf t = Multi
protRef_conf v = _fair    if conf t = Fair
```

`_clean`, `_noth`, `_prim`, `_multi` *and* `_fair` *are references accessing the content of the protection primitives* `Clean`, `Noth`, `Lock`, `Multi` *and* `Fair`. *Lets assume that their implementation is correct, so*

- `m`$_j$ ≡ `get protRef`$_{conf}$ `(sh`$_{conf}$ `m`$_j$`)`
- `sh`$_{conf}$ `x` ≡ `set protRef`$_{conf}$ `x (sh`$_{conf}$ `m`$_j$`)`
- `sh`$_{conf}$ `(f m`$_j$`)` ≡ `update protRef`$_{conf}$ `f (sh`$_{conf}$ `m`$_j$`)`

Theorem 2. `(share, refShare) is a homomorphism for member references`

Proof. If m points to the jth field of the n fields of the constructor, the context of the reference must be a type built by a constructor with n fields: `Ctr m1 ... mj ... mn`

Needed to prove that each operation keeps the homomorphism:

```
get m s ≡ get (refShare m) (share s)
⇔ get m s ≡ get (_j & protRef_conf) (share s)
⇔ mj ≡ get (_j & protRef_conf) (Mem_n (sh_conf m1) ...
                                ... (sh_conf mj) ... (sh_conf mn))
⇔ mj ≡ get _j (get protRef_conf (Mem_n (sh_conf m1) ...
                                ... (sh_conf mj) ... (sh_conf mn)))
⇔ mj ≡ get protRef_conf (sh_conf mj)
```

```
share (set m x s) ≡ set (refShare m) x (share s)
```

Evaluating the right side:

```
set (refShare m) x (share s)
≡ set (_j & protRef_conf) x (share s)
≡ set (_j & protRef_conf) x (Mem_n (sh_conf m1) ...
                                ... (sh_conf mj) ... (sh_conf mn))
≡ update _j (set protRef_conf x) (Mem_n (sh_conf m1) ...
                                 ... (sh_conf mj) ... (sh_conf mn))
≡ Mem_n (sh_conf m1 ... (set protRef_conf x (sh_conf mj)) ... (sh_conf mn))
≡ Mem_n (sh_conf m1) ... (sh_conf x) ...(Mem_n (sh_conf mn))
```

Evaluating the left side:

```
share (set m x s)
≡ share (set m x (Ctr m_1 ... m_j ... m_n))
≡ share (Ctr m_1 ... x ... m_n)
≡ Mem_n (sh_conf m_1) ... (sh_conf x) ... (sh_conf m_n)
```

```
share (update m f s) ≡ update (refShare m) f (share s)
```

Evaluating the right side:

```
update (refShare m) f (share s)
≡ update (_j & protRef_conf) f (share s)
≡ update (_j & protRef_conf) f (Mem_n (sh_conf m_1) ...
                                  ... (sh_conf m_j) ... (sh_conf m_n))
≡ update _j (update protRef_conf f) (Mem_n (sh_conf m_1) ...
                                  ... (sh_conf m_j) ... (sh_conf m_n))
≡ Mem_n (sh_conf m_1) ... (update protRef_conf f (sh_conf m_j)) ...
                      ... (sh_conf m_n)
≡ Mem_n (sh_conf m_1 ... (sh_conf (f m_j)) ... (sh_conf m_n))
```

Evaluating the left side:

```
share (update m f s)
≡ share (update m x (Ctr m_1 ... m_j ... m_n))
≡ share (Ctr m_1 ... (f m_j) ... m_n)
≡ Mem_n (sh_conf m_1) ... (sh_conf (f m_j)) ... (sh_conf m_n)
```

□

Theorem 3. `(share, refShare) is a homomorphism for structural references`

Proof. If the reference `r` is a structural reference, then `r` is a composition of member references: `r` ≡ m_1 `&` ... `&` m_n

From Theorem 1 it is clear that the composition of two homomorpisms will be a homomorphism for references. From Theorem 2 is known that this property holds for member references. This is generalizable by using the fact that the homomorph images (transformed by `refShare`) of structural references will also be structural references. Thanks to that, this is generalizable Theorem 2 to the composition of n member references. □

Definition 11. *Translatable function*

A translatable function is a function f *of type* `A` → `A` *in which only references are used to access elements of type* A.

A translatable function is nearly polymorphic in its argument type because only the use of references constrain the input and output types. This is necessary to be able to change the representation to another form.

Definition 12. *Operating reference*

Lets call the subexpression s *of the body of a function of type* `A` → `A` *operating references iff*

- s *is a reference.*
- s *has the context type of A.*

The operating references have to be changed when the representation of the transformed element is changed.

Definition 13. *Homomorphic translation*
h *is the homomorphic translation of the function* f *of type* `A → A` *iff*

- *h = (h_s, h_r) is a homomorphism from* A *to* B *for all operating references of* f. *See definitions 9 and 12.*
- f *is a translatable function*
- *An operating reference r in f is replaced by (h_r r) inside the translated function.*

It is trivial to prove that the translated function will have the type $B \to B$. The homomorphic translation is done by the programmer using the automatically generated constructs we developed.

Theorem 4. *The homomorphic translation of a function produces the same result.*

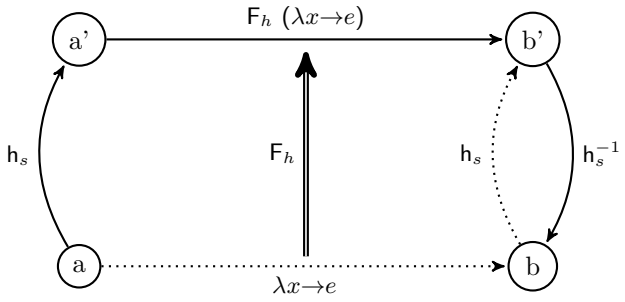

Figure 4: Graphical representation of Theorem 4. a represents the input data. $a' = h_s\ a$ is the homomorph image of the input data, for example, the shared version of the input data. $b = (\lambda x \to e)\ a$ is the result of the computation. $b' = F_h\ (\lambda x \to e)\ a'$.

The result of applying the homomorphic translation of a function to a shared value is the same as applying the original function and sharing the result afterwards: $\mathtt{F}_h\ (\lambda \mathtt{x} \to \mathtt{e}) \circ \mathtt{h}_s \equiv \mathtt{h}_s \circ (\lambda \mathtt{x} \to \mathtt{e})$.

If $\mathtt{h}_s$ *is invertible (this is true in most cases, for example the distribution mapping) also* $\mathtt{h}_s^{-1} \circ \mathtt{F}_h\ (\lambda \mathtt{x} \to \mathtt{e}) \circ \mathtt{h}_s \equiv (\lambda \mathtt{x} \to \mathtt{e})$

Proof. The proof can be easily constructed from the previous theorems.

1. Because f is a translatable function, expressions of type `A` are accessed by references that can be translated.

2. These references could be used with `get`, `set` or `update`, and can be composed with `&`.

3. Because the values and references are translated by a homomorphism, the calls of `get` function evaluates to the same result, while `set` and `update` functions return the homomorphic image of the original result.

4. Subexpressions of `e` that are not operating references remain unchanged.

5. By structural recursion, the result will be the homomorphic image of the original result.

□

Theorem 4 does not take shared constructors into account. The usage of shared constructors is more straightforward because there is a more direct equivalence between the original and shared constructors. So the extension of the theorem to include shared constructors is left to the reader.

7 Case study

Code listings of the case study can be found in the appendix.

We created a case study for our method. We implemented a small application that consists of a number of worker threads, producing log messages and sending them through a channel to a logger thread, or querying log messages, that are sent back on separate channels.

This case study application is implemented using our library, so the configuration that controls the granularity of the paralellization is separated from the representation and the business logic.

We use a simple database representation shown on Listing 1 for the case study. The main loop of the original body is on Listing 2. For the original representation, Feature of the reference package is used to automatically generate structural references as it can be seen on Listing 5.

For the shared version of the case study two configurations had been written (one of them is shown on Listing 4) to enable multiple threads to work on the same database simultaneously. We had to wrap the database into a simple wrapper data type and an IORef [13]. The simple wrapper is needed to let the configuration of the outermost database type. The references for this two data types became the prefixes of our structural references, as it can be seen in the modified main loop on Listing 3. The code did not have to change in any other way. In the implementation we used Haskell type families [14] for configuration.

To measure the overhead caused by the generic definition of the program we manually implemented the same thread-safe logging application by using the concurrency primitives manually. To measure the net gain on thread-safe representation we also implemented the case study with simple central locking.

8 Results

In this paper we demonstrated a method to create a thread-safe representation of an Algebraic Data Type. We inspected how types and values can be converted to this representation. We designed how can the transformation can be controlled by a configuration, and how distributed references and constructor functions can be generated.

We formalized the general concept of homomorphism for references and used it to prove that the shared computation on the shared representation yields the same result.

A simple logging application was implemented as a case study. It had 2 different implementations, one with using concurrency primitives for thread-safety (Manual locking) and the other using our library to generate thread-safe representation (Partial locking). The second approach lets the user to configure the representation so we measured two versions of the case study. One uses Fair nodes for higher levels of the data structure (#1), and the other uses the faster Multi nodes (#2).

The abstraction level of the alternative implementations are measured by the number of effective lines of code, shown on Table 1, and the performance of the implementations is measured by the number of transitions completed in a 2 second interval, seen on Figure 5. Each implementation was tested with different number of read and write operations.

Table 1: Effective lines of code in different implementations of the case study

Implementation	Repr. eloc.	Logic eloc.	Conf. eloc.	Sum eloc.
Manual locking	52	57	0	109
Partial locking	33	54	9	96

On Table 1 it is easy to see that partial locking can be implemented in fewer lines than manual locking. However, the relatively low difference is more important, because it is present in the innermost loops of the application that only contain a few lines. As the manual locking example was implemented we needed to write the functions for handling concurrent lists manually.

The measurements were done on a Lenovo T440p laptop, that has an Intel Core i7™ processor with 16GB of DDR3 RAM. Each measurement was performed 20 times and the results was averaged.

On Figure 5 we can see that all solution that using primitive concurrency operators (Manual locking) does not increase performance more than 25%. Manual

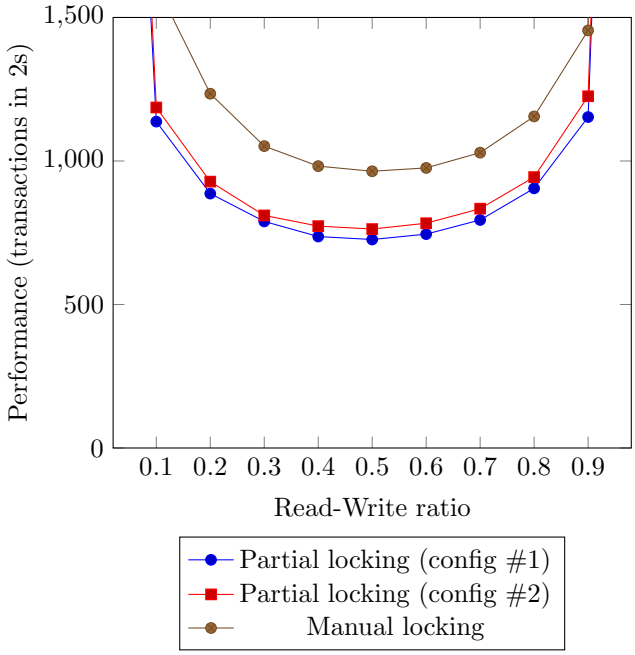

Figure 5: Performance of the case study in different implementations

locking is implemented by using the concurrency primitives of GHC. Using it results a rather complicated and hard-to-maintain code for representation and usage. The better performance of the edge cases is explained by Haskell's lazy evaluation. When the application mostly reads data and never writes, the reads will be much simpler and can be performed quicker than in a balanced situation.

The results show that the library described in this paper gives a higher level of abstraction than the concurrency primitives of GHC, its performance is close to the performance of using the primitives.

8.1 Further work

Our method can be extended by allowing the user to transform bodies of arbitrary functions into a shared format, not only constructors. We would also like to thoroughly investigate properties of references. As part of this we will search for a solution to manage deadlocks and transactions. We intend to consider the correctness of this method when it is generalized to arbitrary functions.

Currently our implementation is based on the GHC generics library [15]. This stops us from using system specific types in shared data types and may cause some performance problems. We would like to experiment with alternatives to the generics library.

References

[1] `http://www.haskell.org/haskellwiki/Algebraic_data_type`

[2] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2011. Data representation synthesis. SIGPLAN Not. 46, 6 (June 2011), 38-49.

[3] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2012. Concurrent data representation synthesis. In Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '12). ACM, New York, NY, USA, 417-428.

[4] Pedro Diniz, Martin Rinard. 1997. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science Volume 1239, 1997, pp 285-299.

[5] Simon Marlow. Parallel and Concurrent Programming in Haskell. 2013. O'Reilly.

[6] Paul Deitel, Harvey Deitel. 2015. Java How To Program, Late Objects Version, Tenth Edition. Prentice Hall. ISBN-10 0132575655

[7] Herbert Schildt. C# 4.0 The Complete Reference. US: McGraw-Hill Osborne Media, 2010.

[8] Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman. 1987. Concurrency Control and Recovery in Database Systems. Addison-Wesley

[9] James E. Burns and Gary L. Peterson. 1987. Constructing multi-reader atomic values from non-atomic values. In Proceedings of the sixth annual ACM Symposium on Principles of distributed computing (PODC '87), Fred B. Schneider (Ed.). ACM, New York, NY, USA, 222-231.

[10] Hackage: references: Generalization of lenses, folds and traversals to handle monads and addition. `http://hackage.haskell.org/package/references`

[11] Philip Wadler. 1992. The essence of functional programming. In Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '92).

[12] Ralf Hinze, Johan Jeuring, Andres Lh. 2007. Comparing Approaches to Generic Programming in Haskell. Lecture Notes in Computer Science. 72-149

[13] John Hughes. 2004. Global variables in Haskell. Journal of Functional Programming. 489-502

[14] `http://www.haskell.org/haskellwiki/GHC/Type_families`

[15] Jos Pedro Magalhes. 2012. The right kind of generic programming. In Proceedings of the 8th ACM SIGPLAN workshop on Generic programming (WGP '12). ACM, New York, NY, USA, 13-24.

Appendix: sample codes from the case study

```
data LogDB = LogDB
  { _criticalErrors :: LogList
  , _errors         :: LogList
  , _debugInfos     :: LogList
  , _startDate      :: Time
  , _lastLogDate    :: Time
  }
data Log = Log
  { _loggerThread   :: String
  , _msg            :: String
  , _loggingDate    :: Time
  }
```

```
data LogList = LogList
  { _logListData :: [Log]
  , _logListNum  :: Size
  }
data Size = Size
  { _sizeInt :: Int
  }
data Time = Time
  { _year    :: Int
  , _month   :: Int
  , _day     :: Int
  , _hour    :: Int
  , _minute  :: Int
  , _sec     :: Int
  }
```

Listing 1: Representation of the case study

```
case q of
  LogThat log →
    do time ← getTime
       (lastLogDate != time
        >=> (debugInfos & logListData !~ (log %:))
        >=> (debugInfos & logListNum & sizeInt !- (+1)))
         logDB

  LogQuery ch →
    do n ← logDB ^! debugInfos & logListNum & sizeInt
       if n > 0 then do
         i ← evalRandIO (fromList (map (,1) [0..n-1]))
         logDB ^?! debugInfos
                     & logListData & distrLogElem (fromIntegral i) & msg
           >>= putMVar ch
       else putMVar ch Nothing
```

Listing 2: Original main loop of the case study's logger

```
case q of
  LogThat log →
    do time ← getTime
       (ioref & wrappedDB & lastLogDate != time
        >=> (ioref & wrappedDB & debugInfos & logListData !~ (log %:))
        >=> (ioref & wrappedDB & debugInfos & logListNum & sizeInt !- (+1)))
         logDB

  LogQuery ch →
    do n ← logDB ^! ioref & wrappedDB & debugInfos & logListNum & sizeInt
       if n > 0 then do
         i ← evalRandIO (fromList (map (,1) [0..n-1]))
         logDB ^?! ioref & wrappedDB & debugInfos
                         & logListData & distrLogElem (fromIntegral i) & msg
          >>= putMVar ch
       else putMVar ch Nothing
```

Listing 3: Main loop of the case study's logger

```
type instance Node LogDB   = Fair
type instance Node LogList = Fair
type instance Node [Log]   = Fair
```

Listing 4: Configuration of the case study representation

```
type instance Node Log     = Prim
type instance Node Time    = Prim
type instance Node Size    = Prim
type instance Node String  = Clean
type instance Node Int     = Clean
type instance Node Char    = Clean
```

```
makeReferences ''Time
makeReferences ''Log
makeReferences ''Size
makeReferences ''LogList
makeReferences ''LogDB
```

Listing 5: Generating the original references

```
makeSharedRefs ''Time
makeSharedRefs ''Log
makeSharedRefs ''LogList
makeSharedRefs ''Size
makeSharedRefs ''LogDB
makeSharedRefs ''LogDBWrapper

makeSharedCons [t| [Log] |]
makeSharedCons ''Log
makeSharedCons ''LogList
makeSharedCons ''LogDB
makeSharedCons ''Time
makeSharedCons ''LogDBWrapper
```

Listing 6: Generating shared references and constructors