Automatic Concurrency Transformations for Query Structures

Anders Kaseorg

May 21, 2015

1 Introduction

The goal of this project is to develop methods for automatic transformation from single-threaded programs to equivalent equivalent concurrent programs, with formally verified preservation of correctness, using the Coq proof assistant.

Specifically, we consider the domain of *query structures*: abstract data types with a predefined collection of methods for queries and modifications. Methods are implemented as programs in a functional language augmented with one or more global mutable *bags* of records supporting SQL-like FIND, INSERT, DELETE, UPDATE operations. The language is intended to be similar to the intermediate language of Fiat [2], so that future work may enable connecting Fiat to the concurrent bag implementations being developed in the Phantom Monitors project [1]. However, this project is self-contained and independent from both of those projects.

2 Domain-specific language

We create a simple domain-specific language for objects with mutable state, such as query structures, based on the functional language Gallina built in to Coq. To augment functional programs with mutating operations, we define the monadic `Program` type paramaterized over an abstract type of instructions and over the type of the program's return value:

```
Inductive Program {Instruction : Type -> Type} {Ret : Type} : Type :=
| Bind {A} : Instruction A -> (A -> Program) -> Program
| Return : Ret -> Program.

Arguments Program : clear implicits.
Notation "x <- f ; m" := (Bind f (fun x => m)) (at level 80, right associativity).
```

The `Instruction` type itself carries no semantics. An instruction set with (nondeterministic) semantics is encapsulated by an `Object` record, which contains the `Instruction` type, a `State` type, and a predicate describing valid state transitions:

```
Record Object := {
    Instruction : Type -> Type;
    State : Type;
    EvalInstruction {A} : Instruction A -> State -> A -> State -> Prop
  }.
```

Given that, we can define the semantics of an entire `Program` as an inductive proposition:

```
Context (obj : Object).

Inductive EvalProgram : forall A,
  Program obj.(Instruction) A -> obj.(State) -> A -> obj.(State) -> Prop :=
| EBind {A i st} {a : A} {st' B f r st''} :
    obj.(EvalInstruction) i st a st' -> EvalProgram B (f a) st' r st'' ->
    EvalProgram B (Bind i f) st r st''
| EReturn {A a st} : EvalProgram A (Return a) st a st.
```

A particular `Object` we are interested in is the bag with FIND, INSERT, DELETE, UPDATE operations. Bags are themselves parameterized over the abstract types: `Item`, which represents one record in the bag; `SearchTerm`, which represents a supported search query (e.g. a SQL WHERE clause); and `UpdateTerm`, which represents a supported modification that can be applied to an `Item` (e.g. a SQL SET clause).

```
Context {Item SearchTerm UpdateTerm : Type}.

Inductive BagInstruction : Type -> Type :=
| IFind : SearchTerm -> BagInstruction (list Item)
| IInsert : Item -> BagInstruction unit
| IDelete : SearchTerm -> BagInstruction (list Item)
| IUpdate : SearchTerm -> UpdateTerm -> BagInstruction (list Item).

Arguments BagInstruction : clear implicits.
```

Given semantics of `SearchTerm` and `UpdateTerm`, we define the semantics of `BagInstruction` (and therefore of `Program BagInstruction`, via `EvalProgram`) in terms of the `list` of `Item`s it contains:

```
Context {searchp : SearchTerm -> Item -> Prop}
        {updatef : UpdateTerm -> Item -> Item}.

Inductive EvalBagInstruction : forall A,
  BagInstruction A -> list Item -> A -> list Item -> Prop :=
  | EFind search items found not_found :
      Partition (searchp search) items found not_found ->
      EvalBagInstruction _ (IFind search) items found items
  | EInsert item items :
      EvalBagInstruction _ (IInsert item) items tt (item :: items)
  | EDelete search items deleted not_deleted :
      Partition (searchp search) items deleted not_deleted ->
      EvalBagInstruction _ (IDelete search) items deleted not_deleted
  | EUpdate search update items affected not_affected :
      Partition (searchp search) items affected not_affected ->
      EvalBagInstruction
        _ (IUpdate search update) items affected
        (map (updatef update) affected ++ not_affected).

Definition bagObject : Object := {|
    Instruction := BagInstruction;
    State := list Item;
    EvalInstruction := EvalBagInstruction
  |}.
```

The DELETE operation returns the list of records that were deleted, and the UPDATE operation returns the list of affected records before modification. The semantics is nondeterministic in that it makes no guarantees about the order in which multiple records might be returned.

To support programs that operate on multiple bags, we define the pairing combinator `pairObject : Object -> Object -> Object` whose `Instruction` type is the sum of two `Instruction` types and whose `State` type is the product of two `State` types, with corresponding semantics.

```
Context (objL objR : Object).

Definition PairInstruction (Ret : Type) := sum (objL.(Instruction) Ret) (objR.(Instruction) Ret).

Definition PairState := prod objL.(State) objR.(State).

Definition EvalPairInstruction A instr st (a : A) st' :=
  match instr with
    | inl instr' => objL.(EvalInstruction) instr' (fst st) a (fst st') /\ (snd st = snd st')
    | inr instr' => objR.(EvalInstruction) instr' (snd st) a (snd st') /\ (fst st = fst st')
  end.

Definition PairObject : Object:= {|
    Instruction := PairInstruction;
    State := PairState;
    EvalInstruction := EvalPairInstruction
  |}.
```

3 A motivating example

The following query structure implements a bank account system in which a nonnegative balance is associated with each account ID. It supports three methods, for account creation, balance inquiry, and transfer between accounts.

```
Definition BankId := nat.
Definition BankBalance := nat.
Record BankAccount := {bank_id : BankId; bank_balance : BankBalance}.
Definition BankSearch := BankId.
Definition BankUpdate := BankBalance -> BankBalance.
Definition bank_searchp id account := account.(bank_id) = id.
Definition bank_updatef u account :=
  {| bank_id := account.(bank_id); bank_balance := u account.(bank_balance) |}.

Inductive BankMethod : Type -> Type :=
| BankCreate : BankId -> BankBalance -> BankMethod bool
| BankInquiry : BankId -> BankMethod BankBalance
| BankTransfer : BankId -> BankId -> BankBalance -> BankMethod bool.

Definition BankImpl {Ret} (m : BankMethod Ret)
: Program (BagInstruction BankAccount BankSearch BankUpdate) Ret :=
  match m with

    | BankCreate id balance =>
      conflicts <- IFind id;
        match conflicts with
          | nil =>
            _ <- IInsert {| bank_id := id; bank_balance := balance |};
              Return true
          | cons _ _ =>
            Return false
        end

    | BankInquiry id =>
      accounts <- IFind id;
        Return
          match accounts with
            | nil => 0
            | cons account _ => account.(bank_balance)
          end

    | BankTransfer source target amount =>
      source_accounts <- IFind source;
        match source_accounts with
          | nil => Return false
          | cons source_account _ =>
            target_accounts <- IFind target;
              match target_accounts with
                | nil => Return false
                | cons target_account _ =>
                  if lt_dec source_account.(bank_balance) amount then
                    Return false
                  else
                    _ <- IUpdate source (fun balance => balance - amount);
                    _ <- IUpdate target (fun balance => balance + amount);
                    Return true
              end
        end

  end.
```

We could formally prove various correctness properties about this bank; for example, the invariant that at any time, the sum of all the current balances associated with accounts in the bank is equal to the sum of all balances that were passed to previous successful `BankCreate` calls.

However, most such properties will only hold in the above single-threaded semantics where each method is run to completion before another method is started. It is obvious that if we were to simply transplant this code into a semantics where multiple methods may be running concurrently, it would be full of race conditions where these properties may be violated.

4 Concurrent semantics

At a high level, we treat a concurrent system as a state machine whose semantics is described by a predicate over its possible transitions under user interactions, where each interaction consists of either spawning a new thread that runs one of the defined methods, or joining one of the previously spawned threads (i.e., waiting for it to return, inspecting its return value, and removing it from the list of threads).

```
Context (Method : Type -> Type) (State : list Type -> Type).

Inductive Interaction : list Type -> Type -> list Type -> Type :=
| Spawn Ret Rets : Method Ret -> Interaction Rets unit (Ret :: Rets)
| Join Ret Rets : Interaction (Ret :: Rets) Ret Rets
| Up Ret Rets Ret' Rets' :
    Interaction Rets Ret' Rets' -> Interaction (Ret :: Rets) Ret' (Ret :: Rets').
```

```
Definition Behavior := forall Rets Ret Rets',
  Interaction Rets Ret Rets' -> State Rets -> Ret -> State Rets' -> Prop.
```

Predicates of this generic type can describe the behavior of a wide variety of different flavors of concurrent systems. In this way we will specify both the concrete implementation, and the abstract specification it should satisfy, with the goal of proving that the concrete `Behavior` refines the abstract `Behavior` under a coinductive simulation relation:

```
Context {Method : Type -> Type}
        {State1 : list Type -> Type}
        (Behavior1 : Behavior Method State1) (* concrete *)
        {State2 : list Type -> Type}
        (Behavior2 : Behavior Method State2) (* abstract *).

CoInductive Simulates Rets (ts1 : State1 Rets) (ts2 : State2 Rets) : Prop :=
| SimStep :
    (forall A Rets' int a ts1',
       Behavior1 Rets A Rets' int ts1 a ts1' ->
       exists2 ts2',
       Behavior2 Rets A Rets' int ts2 a ts2' &
       Simulates Rets' ts1' ts2') ->
    Simulates Rets ts1 ts2.
```

We are interested in the concrete system where each method is implemented as a `Program` over some `obj : Object`; we assume that each individual `obj.(Instruction)` is atomic and serializable (leaving the problem of implementing thread-safe bags to someone else), but instructions from different threads may be interleaved arbitrarily. The state of the entire system is described by a `Program` for each thread representing the future of that thread's computation, along with the global `obj.(State)`.

```
Context (obj : Object)
        {Method : Type -> Type}
        (impl : forall Ret, Method Ret -> Program obj.(Instruction) Ret).

Definition ThreadsState (Rets : list Type) : Type :=
  (fold_right prod unit (map (Program obj.(Instruction)) Rets) * obj.(State))%type.
```

We define a small-step semantics to model the threads' internal computations:

```
Inductive ThreadAdvance :
  forall Rets : list Type ,
    ThreadsState Rets -> ThreadsState Rets -> Prop :=
| TACons Ret Rets p ps st ps' st' :
    ThreadAdvance Rets (ps, st) (ps', st') ->
    ThreadAdvance (Ret :: Rets) ((p, ps), st) ((p, ps'), st')
| TABind Ret Rets ps st A instr f (a : A) st' :
    obj.(EvalInstruction) instr st a st' ->
    ThreadAdvance (Ret :: Rets) ((Bind instr f, ps), st) ((f a, ps), st').

Definition ThreadsAdvance Rets :=
  clos_refl_trans_1n (ThreadsState Rets) (ThreadAdvance Rets).
```

and the effect of each `Interaction` following a period of computation:

```
Inductive ThreadsInteraction :
  forall Rets Ret Rets',
    Interaction Method Rets Ret Rets' -> ThreadsState Rets -> Ret -> ThreadsState Rets' -> Prop :=
| TISpawn Ret Rets m ps st :
    ThreadsInteraction Rets unit (Ret :: Rets) (Spawn _ Ret Rets m)
      (ps, st) tt ((impl Ret m, ps), st)
| TIJoin Ret Rets ret ps st :
    ThreadsInteraction (Ret :: Rets) Ret Rets (Join _ Ret Rets)
      ((Return ret, ps), st) ret (ps, st)
| TICons Ret Rets Ret' Rets' int p ps st ret ps' st' :
    ThreadsInteraction Rets Ret' Rets' int
      (ps, st) ret (ps', st') ->
    ThreadsInteraction (Ret :: Rets) Ret' (Ret :: Rets') (Up _ Ret Rets Ret' Rets' int)
      ((p, ps), st) ret ((p, ps'), st').
```

Combining these gives us a complete `Behavior`:

```
Definition ThreadsBehavior :
  Behavior Method ThreadsState :=
  fun Rets Ret Rets' int ts ret ts'' =>
    exists2 ts' : ThreadsState Rets,
    ThreadsAdvance Rets ts ts' &
    ThreadsInteraction Rets Ret Rets' int ts' ret ts''.
```

Having defined an interleaving semantics for the concrete system, we reuse the same definition over a different `Object` to define the abstract specification. In particular, we build an `Object` in which each `Instruction` represents an entire `Method` rather than an individual bag instruction, with semantics given by `EvalProgram` on that method's implementation. Each thread in the abstract object is then implemented by a single abstract instruction; when that thread is scheduled, it performs the entire method atomically.

```
Context (obj : Object)
        {Method : Type -> Type}
        (impl : forall Ret, Method Ret -> Program obj.(Instruction) Ret).

Definition methodObject : Object := {|
    Instruction := Method;
    State := obj.(State);
    EvalInstruction Ret m := EvalProgram obj Ret (impl Ret m);
  |}.

Definition serialImpl Ret (m : Method Ret) : Program Method Ret :=
  Bind m Return.

Definition SerialBehavior : Behavior Method (ThreadsState methodObject) :=
  ThreadsBehavior methodObject serialImpl.
```

5 Synchronization strategy

In order to support writing useful concurrent programs, we must add some synchronization primitives as extra instructions. We will start by assuming the existence of a family of arbitrarily many locks (parameterized over some index type), supporting ACQUIRE and RELEASE operations.

The goal is then to start with a single-threaded query structure, such as the bank example above, and decide which ACQUIRE and RELEASE instructions to insert such that any correctness property of the single-threaded code carries over to the new concurrent code.

One strategy for doing this that works in simple cases, including the bank example, is as follows.

1. Analyze each method to find a predicate satisfied by all records that the method touches, by taking the disjunction of all search predicates used in bag operations. (For UPDATE operations, we need to make sure that both the old and new versions of the affected records satisfy the predicate.) In the bank example, the result would be `bank_id = id` for `BankCreate id balance`, `bank_id = id` for `BankInquiry id`, and `bank_id = source \/ bank_id = target` for `BankTransfer source target amount`.

2. Create a family of locks corresponding to a family of disjoint predicates that tiles the disjunctive components of the predicates above. In the bank example, we could create a lock corresponding to `bank_id = id` for each `id`.

3. Create a total ordering on this family.

4. For each method, find a set of locks whose predicates cover all the records that the method touches. This should be a finite set (otherwise this strategy has failed).

5. At the beginning of the method, insert code to sort its set of locks according to the total ordering (to prevent an AB-BA deadlock), followed by an ACQUIRE instruction for each lock in this order.

6. Before each RETURN instruction in the method, insert a RELEASE instruction for each of that method's locks (in any order).

In the current implementation, the locking scheme must be constructed manually; its soundness is then verified by an automated proof. For example, the bank example is supported by the following manually defined locking scheme:

```
Inductive BankLock :=
| BankLockAccount : BankId -> BankLock.

Definition BankLockOwns account lock : Prop :=
  match lock with
    | BankLockAccount id => account.(bank_id) = id
  end.

Definition BankMethodLocks {Ret} (m : BankMethod Ret) : list BankLock :=
  match m with
```

```
    | BankCreate id _ => [BankLockAccount id]
    | BankInquiry id => [BankLockAccount id]
    | BankTransfer source target _ =>
      if lt_dec source target
      then [BankLockAccount source; BankLockAccount target]
      else [BankLockAccount target; BankLockAccount source]
  end.
```

The automatic verification is designed with the goal of eventually facilitating automation of more steps of locking scheme construction; that will be the subject of future work.

6 Synchronization implementation

We implement the family of mutexes as a separate `Object`.

```
Context (Mutex : Type).

Inductive MutexInstruction : Type -> Type :=
| Acquire : Mutex -> MutexInstruction unit
| Release : Mutex -> MutexInstruction unit.

Definition MutexState := list Mutex.

Inductive EvalMutexInstruction :
  forall A, MutexInstruction A -> MutexState -> A -> MutexState -> Prop :=
| EAcquire mutex held :
    ~ In mutex held ->
    EvalMutexInstruction unit (Acquire mutex) held tt (mutex :: held)
| ERelease held mutex held' :
    EvalMutexInstruction unit (Release mutex) (held ++ mutex :: held') tt (held ++ held').

Definition mutexObject : Object := {|
    Instruction := MutexInstruction;
    State := MutexState;
    EvalInstruction := EvalMutexInstruction
  |}.
```

Combining this with another `Object` using the `pairObject` combinator gives us the full instruction set targeted by the concurrency transformation.

```
Context (obj : Object)
        (Method : Type -> Type)
        (impl : forall Ret, Method Ret -> Program obj.(Instruction) Ret)
        (Mutex : Type)
        (Owns : obj.(Item) -> Mutex -> Prop)
        (methodLocks : forall Ret, Method Ret -> list Mutex).

Definition twoPhaseObject := pairObject obj (mutexObject Mutex).

Fixpoint acquireLocks {Ret} (locks : list Mutex) (p : Program twoPhaseObject.(Instruction) Ret) :
  Program twoPhaseObject.(Instruction) Ret :=
  match locks with
    | nil => p
    | lock :: locks' => _ <- inr (Acquire Mutex lock); acquireLocks locks' p
  end.

Fixpoint releaseLocks {Ret} (locks : list Mutex) (p : Program twoPhaseObject.(Instruction) Ret) :
  Program twoPhaseObject.(Instruction) Ret :=
  match locks with
    | nil => p
    | lock :: locks' => releaseLocks locks' (_ <- inr (Release Mutex lock); p)
  end.

Definition twoPhaseImpl Ret (m : Method Ret) : Program twoPhaseObject.(Instruction) Ret :=
  let locks := methodLocks Ret m in
  acquireLocks
    locks
    (foldProgram
       (fun _ instr => Bind (inl instr))
       (fun ret => releaseLocks locks (Return ret))
       (impl Ret m)).
```

7 Correctness theorem

We wish to prove that the behavior of the concurrent implementation refines the behavior of the abstract specification under the `Simulates` relation:

```
Definition abstractObject := methodObject obj impl.

Definition TwoPhaseCorrectness (st : obj.(State)) : Prop :=
  Simulates (ThreadsBehavior twoPhaseObject twoPhaseImpl)
            (SerialBehavior abstractObject)
            nil
            (tt, (st, nil))
            (tt, st).
```

Before proving this, we need to be able to reason about the working sets of our instructions such that two instructions with disjoint working sets commute with each other. We add some extra fields to the `Object` record for this purpose:

```
Record Object :=
  {
    Instruction : Type -> Type;
    State : Type;
    EvalInstruction {A} : Instruction A -> State -> A -> State -> Prop;
    Item : Type;
    Touches {A} : Instruction A -> (Item -> Prop) -> Prop;
    commute {A B} instrA instrB P Q :
      Touches instrA P ->
      Touches instrB Q ->
      (exists2 item, P item & Q item) \/
      (forall st (a : A) st' (b : B) st'',
         EvalInstruction instrA st a st' ->
         EvalInstruction instrB st' b st'' ->
         exists2 st''', EvalInstruction instrB st b st''' & EvalInstruction instrA st''' a st'')
  }.
```

Now we need to define a correspondence between concrete states and abstract states that is preserved by each step of execution. This works as follows. Each concrete thread has a “commit point” before its first RELEASE instruction. For each thread that has not yet passed its commit point, we imagine updating the abstract state by applying all the instructions that thread has already executed, asserting that the working sets of those instructions are covered by the locks previously acquired by the thread.

```
Fixpoint Any {A} (P : A -> Prop) (l : list A) : Prop :=
  match l with
    | nil => False
    | cons x xs => P x \/ Any P xs
  end.

Inductive PreCommit Ret :
  Method Ret -> twoPhaseObject.(State) -> Program TwoPhaseInstruction Ret -> list Mutex -> twoPhaseObject.(State) -> Prop :=
| PreImpl m st :
    PreCommit Ret m st (twoPhaseImpl Ret m) nil st
| PreInstr m A instr f locks (a : A) st st' st'' :
    obj.(Touches) instr (fun item => Any (Owns item) locks) ->
    PreCommit Ret m st (Bind (inl instr) f) locks st' ->
    twoPhaseObject.(EvalInstruction) (inl instr) st' a st'' ->
    PreCommit Ret m st (f a) locks st''
| PreAcquire m lock f locks st st' st'' :
    PreCommit Ret m st (Bind (inr (Acquire _ lock)) f) locks st' ->
    twoPhaseObject.(EvalInstruction) (inr (Acquire _ lock)) st' tt st'' ->
    PreCommit Ret m st (f tt) (lock :: locks) st''.
```

For each thread that has passed its commit point, we imagine updating the concrete state by applying all the instructions that thread has not yet executed.

```
Inductive PostCommit Ret :
  Program TwoPhaseInstruction Ret -> list Mutex -> twoPhaseObject.(State) -> Ret -> twoPhaseObject.(State) -> Prop :=
| PostReturn ret locks st :
    PostCommit Ret (Return ret) locks st ret st
| PostRelease lock locks f ret st st' st'' :
    twoPhaseObject.(EvalInstruction) (inr (Release _ lock)) st tt st' ->
    PostCommit Ret (f tt) locks st' ret st'' ->
    PostCommit Ret (Bind (inr (Release _ lock)) f) (lock :: locks) st' ret st''.
```

Observe that it should not matter which order we process the threads in, because all of the imaginary updates to the concrete state commute with each other, and all of the imaginary updates to the abstract state commute with each other—hence we use the arbitrary order in which the threads are already listed. After doing this with all threads, we are imagining a new concrete state and a new abstract state. If these states turn out to be the same, then we say that the original concrete state corresponds to the original abstract state.

```
Inductive TwoPhaseMapState' :
  forall Rets,
    ThreadsState twoPhaseObject Rets ->
    fold_right prod unit (map (Program Method) Rets) ->
    twoPhaseObject.(State) ->
    Prop :=
| TPMSNil st :
    TwoPhaseMapState' nil (tt, st) tt st
| TPMSPre Ret Rets p1 locks ps1 st1 m ps2 st2 st2' :
    PreCommit Ret m st2 p1 locks st2' ->
    TwoPhaseMapState' Rets (ps1, st1) ps2 st2' ->
    TwoPhaseMapState' (Ret :: Rets) ((p1, ps1), st1) (Bind m Return, ps2) st2
| TPMSPost Ret Rets p1 locks ps1 st1 st1' ret ps2 st2 :
    PostCommit Ret p1 locks st1 ret st1' ->
    TwoPhaseMapState' Rets (ps1, st1') ps2 st2 ->
    TwoPhaseMapState' (Ret :: Rets) ((p1, ps1), st1) (Return ret, ps2) st2.

Definition TwoPhaseMapState Rets (ts1 : ThreadsState twoPhaseObject Rets) (ts2 : ThreadsState abstractObject Rets) : Prop :=
  let (ps2, st2) := ts2 in TwoPhaseMapState' Rets ts1 ps2 (st2, nil).
```

The Coq proof that this correspondence is preserved by each step of execution is still a work in progress with many `admit`s:

```
Lemma advance
      Rets ts1 ts1' ts2
      (ms : TwoPhaseMapState Rets ts1 ts2)
      (ta1 : ThreadAdvance twoPhaseObject Rets ts1 ts1') :
  exists2 ts2',
  ThreadsAdvance abstractObject Rets ts2 ts2' &
  TwoPhaseMapState Rets ts1' ts2'.

Lemma interact
      Rets Ret Rets' ts1 a ts1' ts2
      (ms : TwoPhaseMapState Rets ts1 ts2)
      (int : Interaction Method Rets Ret Rets')
      (ti : ThreadsInteraction twoPhaseObject twoPhaseImpl Rets Ret Rets' int ts1 a ts1') :
  exists2 ts2',
  ThreadsInteraction abstractObject (serialImpl abstractObject) Rets Ret Rets' int ts2 a ts2' &
  TwoPhaseMapState Rets' ts1' ts2'.
```

From these unfinished lemmas, we deduce the theorem we are trying to prove with no further `admit`s:

```
Theorem TwoPhaseCorrect st : TwoPhaseCorrectness st.
```

8 “Conclusion”

My project started with several very ambitious goals, and it’s disappointing to have made only partial progress toward some of them over the semester. But at least it has been fun in a masochistic sort of way, and I expect to continue working on it over the summer. Hopefully then I will have a more interesting conclusion to write.

References

[1] Christian J. Bell, Mohsen Lesani, Adam Chlipala, Gregory Malecha, Stephan Boyer, and Peng Wang. Phantom monitors: A simple foundation for modular proofs of fine-grained concurrent programs. 2015. (to appear).

[2] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’15, pages 689–700, New York, NY, USA, 2015. ACM.