

Seminar: Programming models and Code Generation
Winter Term 2014/2015
Quasi deterministic parallel programming using LVars

Nishanth Nagendra
Technische Universität München

15.01.2015

Abstract

A *deterministic program* always produces the same observable result on multiple runs and a *deterministic by construction parallel programming model* allows only such programs to be written offering programmers freedom from hard to reproduce non deterministic bugs. This report talks about *Lvish*, a *quasi deterministic parallel programming model* that extends *LVars*, its predecessor, which is a deterministic model. *LVars* generalize existing single assignment models by allowing communication through shared monotonic data structures with monotonic write operations and threshold read operations that block until a lower bound is reached. The semantics are defined in terms of an application specified lattice. *LVish* extends *LVar* by allowing to freeze them in order to read their exact contents and to attach event handlers to an *LVar* triggering a callback when their state changes. It offers quasi determinism in the worst case which means on every run it either produces the same result or raises an error. The *LVish* model yields promising parallel speedup on benchmarking applications using a prototype implementation of a library in haskell.

1 Introduction

The purpose of a deterministic-by-construction programming model is to rule out programs like the

following from being written, or force them to behave deterministically. Written in a hypothetical parallel language.

```
let _ = put lv 3 in /write 3 in lv/  
let par v = get lv /read from lv/  
        _ = put lv 4  
/write 4 in lv/  
in v /final value is v/
```

The *let par* expression, evaluates the two sub-expressions in parallel. The value of *v* is the value of the entire program. Depending on the scheduler *get lv* or *put lv4* can run first, *v* might be either 3 or 4 during multiple runs giving rise to non determinism.

A popular approach for enforcing determinism is to require that variables can only be assigned to once, giving us a single-assignment language. Our program would be invalid because it tries to write to *lv* twice. Single-assignment variables with blocking read semantics, often known as *IVars*, turn up in all kinds of deterministic parallel systems. But the single-assignment rule disallows a number of programs that we would like to permit such as writing the same value to an *IVar* twice. As another example, consider the problem of filling in various parts of a data structure in parallel. Say that *lv* contains a pair, and we are writing to *lv* twice, with each write filling in one component of the pair.

```
let par _ = put 1 (4, )  
        _ = put 1 ( , 3) /error/
```

in get l

Read from *lv* is possible once both writes have completed, so it will always be (4, 3). But single-assignment rules out this program (although we could fake it by using a distinct IVar for each component of the pair, but with more sophisticated data structures it becomes difficult to fake a solution with IVars.)

There is a need to generalize the single-assignment model to something that admits programs like these but retains determinism. Deterministic algorithms must be expressible in a style that guarantees determinism, and non-deterministic behaviors, where desired, must be requested explicitly. More generally, enforcing deterministic semantics simplifies composing, porting, reasoning about, debugging, and testing parallel software. Language-level enforcement of determinism is possible, but existing models tend to lack features that would make them applicable to a broader range of problems. Moreover, they lack extensibility.

Existing models A number of models currently exist as a part of the long running effort towards deterministic by construction parallel programming.

- *No-shared-state parallelism*: Pure functional programming in Haskell with `par` and `pseq` combinators is an example. It uses function level task parallelism or futures. This kind of parallelism does not allow for any shared mutable state between tasks, forcing tasks to produce values independently. Models based on pure data parallelism like *Data parallel Haskell*, *River Trail API for Java script* also fall in this category,
- *Data-flow parallelism*: Kahn process networks, synchronous data flow systems are examples. In these models, tasks can only communicate via channels which are FIFO queues where the read operation on these queues is blocking until data is available. Stream-processing languages such as *StreamIt* are based on KPN.,

- *Single-assignment parallelism*: IVars are an example which are memory locations that can be either full or empty and can only be assigned once. Reads are blocking for IVars. They have been used in the *monad-par* Haskell library, *Intel CnC* (Concurrent collections, Concurrent ML (as *SnycVars*)).,
- *Imperative-disjoint parallelism*: The state accessed by concurrent threads are disjoint. *Deterministic Parallel Java* falls under this category.

A common theme that emerges in the study of deterministic-by-construction systems is the notion of monotonicity. Many deterministic parallel programs cannot be expressed with these models. Sharply restricted communication and synchronization capabilities have consequences not only for the immediate usability of guaranteed-deterministic languages, but for the extensibility of those languages as well.

Motivating Example Consider applications in which independent sub-computations contribute information to shared data structures that are unordered, irregular, or application-specific. Following is an example that highlights the issues of ordering which create a friction between parallelism and determinism.

In a directed graph where *v* represents a particular node, we wish to find all (or the first *k* degrees of) node connected to *v*, then analyze each node in that set. Existing parallel solutions [1] often use a non-deterministic traversal of the connected component even though the final connected component is deterministic. For example, IVars cannot accumulate sets of visited nodes, nor can they be used as mark bits on visited nodes, since they can only be written once and not tested for emptiness. Streams, on the other hand, impose an excessively strict ordering for computing the unordered set of vertex labels in a connected component. A purely functional attempt is given in the original paper [1] where connected-component discovery is parallel, but members of the output set do not become avail-

able to other computations until component discovery is finished, limiting parallelism. An LVars based programming model can possibly allow us to write a breadth first search traversal where threads communicate using shared monotonic data structures. Consumers of the data structure may execute as soon as data is available, but may only observe irrevocable, monotonic properties of it.

2 LVars and LVish

2.1 LVars

An LVar is a memory location that can be shared between multiple threads whose contents can only grow bigger over time, for some definition of bigger that the user of the LVar gets to specify.

LVars, short for lattice variables, are a generalization of IVars, which are variables that can only be written to once. (The I is for immutable.) LVars, on the other hand, allow multiple writes, as long as those writes are monotonically increasing with respect to a user-specified lattice. Meanwhile, LVar reads have to specify a threshold set of possible states. A read will block until one of the states in the threshold set has been reached, and then return that state. Information can never be removed and the order in which it is added is not observable. IVars turn out to be a special case of LVars. The property of monotonically increasing write operations is enforced by the language. The semantics of *put* says that a write to a location is the least upper bound of the new value and the current value.

```
let par _ = put num 3
      _ = put num 2
in get num /always 3/
```

Monotonic writes alone aren't enough to ensure deterministic programs. The above program is non-deterministic where num's notion of bigger is \leq . Despite writes always being monotonically increasing. Get might read either 2 or 3, depending on the order of *put*/*get* operations determined by the scheduler.

- To maintain determinism, *get* operation is restricted: instead of reading the LVars exact value, we read one of a set of lower bounds on its value that are given as an extra argument to *get*. The *get* operation will block until the LVar reaches a value that is at or above one of those lower bounds. For the above example, the threshold set will just be $\{3\}$. *Get* operation is called a threshold read, because the value it returns is the threshold we have crossed.

- Once num's value it is at least 3, it will remain at or above 3 forever. As long as we only access them through the *put*/*get* interface, LVars are thread-safe: we can safely share them between threads without introducing unintentional non-determinism. That is, *puts* and *gets* can happen in any order, without changing the value that a program evaluates to.

The resulting model is general enough to subsume the existing single-assignment model. The user-specified set just needs to be a *join-semilattice*, which is a partially ordered set in which every two elements have a least upper bound.

To describe Lvar formally, a parallel call-by-value λ -calculus named λ_{LVar} is introduced in the paper [1] which is extended with a *store* and communication primitives *put* and *get* that operate on data in the *store*. The definition of λ_{LVar} is parameterized by the choice of lattice and therefore λ_{LVar} is actually a family of languages, rather than a single language. An LVar may have an arbitrary number of states forming a set D , which is partially ordered by a relation \sqsubseteq . Formally, D is a *bounded-join-semilattice* augmented with a greatest element \top .

- D has a least element \perp , representing the initial empty state of an Lvar,
- D has a greatest element \top , representing the error state that results from conflicting updates to an Lvar,

- D comes equipped with a partial order \sqsubseteq , where $\perp \sqsubseteq d \sqsubseteq \top$ for all $d \in D$.
- Every pair of elements in D has a least upper bound (lub) written \sqcup which means it is possible for two sub-computations to independently update an LVar, and then deterministically merge the results by taking the lub of the resulting two states.

Data structures like arrays, tree, maps, streams etc. can be represented as lattice. Figure 2 gives three examples of lattices for common data structures.

During the evaluation of a λ_{LVar} program, a store S keeps track of the states of LVars. Each LVar is represented by a *binding* from a location l , drawn from a countable set Loc , to its *state*, which is some element $d \in D$. Although each LVar in a program has its own state, the states of all the LVars are drawn from the same lattice D . We can do this with no loss of generality because lattices corresponding to different types of LVars could always be unioned into a single lattice (with shared \perp and \top Elements).

Definition 1: A *store* is either a finite partial mapping $S : Loc \xrightarrow{\text{fin}} (D - \{\top\})$, or the distinguished element \top_S .

The state space of *stores* forms a *bounded-join-semilattice* augmented with a greatest element, just as D does, with the empty store \perp_S as its least element and \top_S as its greatest element. The \sqsubseteq and \sqcup operations defined on elements of D can be lifted to the level of stores.

Definition 2: A store S is *less than or equal* to a store S' (written $S \sqsubseteq_S S'$) iff:

- $S' = \top_S$, or
- $\text{dom}(S) \subseteq \text{dom}(S')$ and for all $l \in \text{dom}(S)$, $S(l) \sqsubseteq S'(l)$.

Definition 3: The lub of stores S_1 and S_2 (written $S_1 \sqcup_S S_2$) is defined as follows:

- $S_1 \sqcup_S S_2 = \top_S$ iff there exists some $l \in \text{dom}(S_1) \cap \text{dom}(S_2)$ such that $S_1(l) \sqcup S_2(l) = \top$.
- Otherwise, $S_1 \sqcup_S S_2$ is the store S such that:
 - $\text{dom}(S) = \text{dom}(S_1) \cup \text{dom}(S_2)$, and
 - For all $l \in \text{dom}(S)$:

$$S(l) = \begin{cases} S_1(l) \sqcup S_2(l) & \text{if } l \in \text{dom}(S_1) \cap \text{dom}(S_2) \\ S_1(l) & \text{if } l \notin \text{dom}(S_2) \\ S_2(l) & \text{if } l \notin \text{dom}(S_1) \end{cases}$$

By Definition 3, if $d_1 \sqcup d_2 = \top$, then $[l \mapsto d_1] \sqcup_S [l \mapsto d_2] = \top_S$. Notice that a store containing a binding $l \mapsto \top$ can never arise during the execution of a λ_{LVar} program, because an attempted write that would take the state of l to \top would raise an error before the write can occur.

Communication Primitives The new, put, and get operations create, write to, and read from LVars, respectively.

- *new* extends the store with a binding for a new LVar whose initial state is \perp , and returns the location l of that LVar (i.e. a pointer to the LVar).
- *put* takes a pointer to an LVar and a singleton set containing a new state and updates the LVar's state to the least upper bound of the current state and the new state, potentially pushing the state of the LVar upward in the lattice. Any update that would take the state of an LVar to \top results in an error.
- *get* performs a blocking threshold read. It takes a pointer to an LVar and a *threshold set* Q , which is a non-empty subset of D that is *pairwise incompatible*, meaning that the lub of any two distinct elements in Q is \top . If the LVar's state d_1 in the lattice is *at or above* some $d_2 \in Q$, the *get* operation unblocks and returns d_2 . Note that d_2 is a unique element of Q , for if there is another $d'_2 \neq d_2$ in the threshold set such that $d'_2 \sqsubseteq d_1$, it would follow that $d_2 \sqcup d'_2 \sqsubseteq d_1$, and so $d_2 \sqcup d'_2$ cannot be \top , which contradicts the requirement that T be pairwise incompatible.

For Lambda Lvar syntax and semantics, remaining definitions and its details please refer XXX. Frame rule figure should come here.

Proof of determinism: The key to the determinism is the frame rule given in *Figure 2*. A frame property captures the idea of local reasoning about programs that alter state. Here, C is a program, and $\{p\}C\{q\}$ says that if the assertion p is true before C runs, then the assertion q will be true afterwards. For example, p and q might describe the state of the heap before and after it is updated by C . It tells that running C starting from a state satisfying the assertion $p \wedge r$ will result in a state satisfying the assertion $q \wedge r$. The assertion $p * r$ is satisfied by a heap if the heap can be split into two non-overlapping parts satisfying p and r , respectively. For LVars, the frame property says that independent effects commute with each other. Consider an expression e that runs starting in store S and steps to the expression e , updating the store to S' . Frame property guarantees that if e starts from a larger store $S \sqcup R$, where R is some other store *framing on* to S , then e will update the store to $S' \sqcup R$, and e will step to e . The *frame store* R is a store resulting from some other independently-running computation. S and R (or S' and R) need not be disjoint — we are combining them with \sqcup , not with \wedge . If they did have to be disjoint, it would mean that concurrent operations cannot touch the same parts of the store. For LVars, this kind of total disjointness is unnecessary, since updates commute! Instead, we can behave as though S and R are disjoint, even though they might not be.

Diamond Lemma: This does the heavy lifting of the determinism proof: it establishes the diamond property, which says that if a configuration steps to two different configurations, there exists a single third configuration to which those configurations both step. For supporting lemmas, its details and complete proofs please refer to XXXXXXXXXX. Lemmas there also show how cases of deadlocks and livelocks are also handled.

The LVars model guarantees determinism by

monotonic write operations [which can commute] and threshold read operations. But it is not as general-purpose as one might hope. Consider an unordered graph traversal. A typical implementation involves a monotonically growing set of *seen nodes*; neighbors of seen nodes are fed back into the set until it reaches a fixed point. These are not expressible using the threshold read and least-upper-bound write operations described above. The problem is that these computations rely on negative information about a monotonic data structure, i.e., on the absence of certain writes to the data structure. In a graph traversal, neighboring nodes should only be explored if the current node is not yet in the set; a fixpoint is reached only if no new neighbors are found; and at the end of the computation it must be possible to learn exactly which nodes were reachable. But in the LVars model, asking whether a node is in a set means waiting until the node is in the set, and it is not clear how to lift this restriction while retaining determinism.

2.2 LVish

In this regard, LVish extends the Lvars model with 2 additional capabilities. First, it adds event handlers, a mechanism for attaching a callback function to an LVar that runs, asynchronously, whenever events arrive (in the form of monotonic updates to the Lvar). Second, a primitive for freezing an LVar, which means: once an LVar is frozen, any further writes that would change its value instead throw an exception; on the other hand, it becomes possible to discover the exact value of the LVar, learning both positive and negative information about it, without blocking. Freezing does not commute with writes. If a freeze is interleaved before such a write, the write will raise an exception; if it is interleaved afterwards, the program will proceed normally. Although it appears like determinism is lost, these programs satisfy the property of Quasi determinism that is they will always output the same result or an error.

`addHandler lv {1,3,5,...} (λx.put lv x + 1)` (1)

This registers a handler for lv that executes the callback function $\lambda x. put\ lv\ x + 1$ for each odd number that lv is at or above. Event sets are a mathematical modeling tool only. Event handlers in LVish invoke their callback for all events in their event set Q that have taken place (i.e., all values in Q less than or equal to the current LVar value), even if those events occurred prior to the handler being registered. If a callback only executed for events that arrived after its handler was registered, or only for the largest event in its handler set that had occurred, then it could result in non-determinism. For example if puts are interleaved with handler registration as a part of a par expression then a get operation outside this would block, or not, depending on how the handler registration was interleaved with the puts. By instead executing a handlers callback once for each and every element in its event set below or at the LVars value, quasi-determinism is guaranteed.

Quiescence through handler pools: Event handlers are asynchronous, so a separate mechanism is needed to determine when they have reached a quiescent state, i.e., when all callbacks for the events that have occurred have finished running. Detecting quiescence is crucial for implementing fixpoint computations. To build flexible data-flow networks, it is also helpful to be able to detect quiescence of multiple handlers simultaneously. This design includes handler pools, which are groups of event handlers whose collective quiescence can be tested.

```
let h = newPool
in addInPool h lv Q f;
quiesce h
```

The quiesce operation blocks until all of the callbacks running in a particular handler pool, in this case h , are done running. where lv is an LVar, Q is an event set, and f is a callback. Handler pools are created with the *newPool* function, and handlers are registered with *addInPool*. Quiescence of a handler is a non-monotonic property: it can move in and out of quiescence as more puts to an LVar occur. There is no risk to quasi-determinism,

however, because quiesce does not yield any information about which events have been handled, any such questions must be asked through Lvar *get*.

The freeze operation is a non-blocking read that lets us find out the exact contents of an LVar. After freezing, any further writes that would change LVar's state will raise an exception! A safe practice is to quiesce before freezing. For a program that performs freezes, Only two possible outcomes are guaranteed: the deterministic result of all the effects, or a write-after-freeze exception that can assist in debugging the synchronization bug. This property is called quasi-determinism, and the LVish model is quasi-deterministic.

Freezing introduces quasi-determinism to the programming model, because we might freeze before we have quiesced, or because our thread that's freezing and quiescing might be racing with some other thread to write into a shared LVar. But, if it is ensured that freezes only ever happen after all writes have completed, then the computations will be deterministic, because there wouldn't be any write-after-freeze races. This is enforced at the implementation level which takes care of quiescing and freezing at the end of a computation. In the LVish library, this is provided by the *runParThenFreeze* function.

[Paper] contains a quasi-deterministic, parallel, call-by-value λ_{LVish} -calculus extended with a store containing LVars. It extends the original LVar formalism to support event handlers and freezing. Rather than modeling the full ensemble of event handlers, handler pools, quiescence, and freezing as separate primitives, it instead formalizes the *freeze-after* pattern which is a simpler primitive combining all others.

An LVars state is now a pair (d, frz) , where d is an element of the application-specific set D and frz (Short for freeze) is a status bit of either *true* or *false*. We define an ordering \sqsubseteq_p on LVar states (d, frz) in terms of the application-specific ordering \sqsubseteq on elements of D . Every element of D is “freezable” except \top . Informally:

- Two unfrozen states are ordered according to the application-specific \sqsubseteq ; that is,

$(d, false) \sqsubseteq_p (d', false)$ exactly when $d \sqsubseteq d'$.

- Two frozen states do not have an order, unless they are equal: $(d, true) \sqsubseteq_p (d', true)$ exactly when $d = d'$.
- An unfrozen state $(d, false)$ is less than or equal to a frozen state $(d', true)$ exactly when $d \sqsubseteq d'$.
- A frozen state is less than an unfrozen state if the unfrozen state is \top ; that is, $(d, true) \sqsubseteq_p (d', false)$ exactly when $d' = \top$.

The addition of status bits to the application-specific lattice results in a new lattice $(D_p, \sqsubseteq_p, \perp_p, \top_p)$. \sqcup_p for the lub operation that \sqsubseteq_p induces. During the evaluation of LVish programs, a *store* S keeps track of the states of LVars and represents them by a binding from a location l , drawn from a set loc , to its state, which is some pair (d, frz) from the set D_p . For detailed syntax and semantics of LVish please refer to the paper[7].

Communication primitives: They are same as Lvar except that new will now create a binding for a new Lvar whose initial state is $(?, false)$. Put takes a pointer to the Lvar and a new lattice element which is a pair. Get similarly takes a threshold set containing pairs. *freeze e_{lv} after e_{events} with e_{cb}* has the following semantics: It attaches the callback e_{cb} to the LVar e_{lv} . The expression e_{events} must evaluate to a event set Q ; the callback will be executed, once, for each lattice element in Q that the LVars state reaches or surpasses. The callback takes a lattice element as its argument. Its return value is ignored, so it runs solely for effect. For instance, a callback might itself do a put to the LVar to which it is attached, triggering yet more callbacks. If the handler reaches a quiescent state, the LVar e_{lv} is frozen, and its exact state is returned (rather than an under-approximation of the state, as with *get*).

To keep track of the running callbacks, LVish includes an auxiliary form of the above pattern

taking additional arguments: the set of running callbacks B and a set of values H in the event set for which callbacks have already been launched. It detects quiescence by checking: Every event of interest that has occurred must be handled (be in H). Second, all existing callback threads must have completed and terminated with a value. If yes, then LVars state is frozen. To ensure quiescence is a permanent state, rather than a transient one, freezing is usually the very last step of an algorithm, permitting its result to be extracted. The *runParThenFreeze* function inside LVish library does this, and thereby guarantees full determinism.

Proof of Quasi determinism for LVish:

According to the paper[7] the main theorem says that if two executions starting from a configuration σ terminate in configurations σ' and σ'' , then either σ' and σ'' are the same configuration (up to a permutation on locations), or one of them is *error*.

Independence property: Like LVars, the LVish model uses the frame property again to capture the idea that independent effects commute with each other but this time with some restrictions. If we know that a *store* S evaluating expression e steps to a new *store* S' with e' then it is valid to say that $S \sqcup_S S''$ evaluating e steps to a new store $S' \sqcup_S S''$ with e' if the following conditions are satisfied:

- The status bits of $S' \sqcup_S S''$ and S agree with each other for all locations shared between them.
- Locations in S'' cannot share the same name with locations newly allocated during the transition from one configuration to another.

Please refer to the original paper and the technical report for elaborate details on lemmas and their proofs.

3 Implementation and Performance

A prototype implementation of LvisH is available as a monadic library in Haskell at:

<http://hackage.haskell.org/package/lvish>

It adopts the basic approach of the `par` monad. The existing `Par` monad[3] in Haskell offers an API for parallel programming. It provides by default a work-stealing scheduler and supports forking tasks. The programmer specifies how information flows across computations but not the order in which they will be evaluated at runtime. Information flow is described using `IVars`. With `LVish` we get a lattice-generic infrastructure: the `Par` monad itself, a thread scheduler, support for blocking and signaling threads, handler pools, and event handlers. This infrastructure does not guarantee quasi-determinism, only data structure authors should import it who will implement a specific monotonic data structure using this library providing efficient parallel access, exporting a limited interface for its users ensuring quasi-determinism. `LVish` library provides its own custom scheduler and programs run only inside the `par` monad allowing only `Lvish`-sanctioned side effects to provide compile time guarantees of determinism and quasi determinism.

In order to specify the type of the `par` computation it provides a phantom type that indicates the determinism level.

```
data Determinism = Det | QuasiDet
```

`Par` type constructor:

```
Par :: Determinism -> * -> *
```

following suite of run functions

```
runPar :: Par Det a -> a\\
```

```
runParIO :: Par l v l a -> IO a
runParThenFreeze :: Deep frz a => \\
  Par Det a -> FrzType a
```

If the `Lvish` code uses `freeze` then it is marked as `QuasiDet` or `Det` if it is fully deterministic. *runParIO* allows us to execute `Lvish` code in the `IO` monad if the determinism level is arbitrary. *runParThenFreezes* automatically freezes the `LVar` when it returns back its exact value guaranteeing determinism.

Key Ideas: *Atoms and Idempotence:* Atoms are elements not equal to \perp but whose only smaller element is \perp . Lattices for which every element is the lub of some set of atoms are called atomistic, and most application-specific lattices used by `LVish` programs have this property and the corresponding data structure usually exposes operations that work at the atom level, semantically limiting puts to atoms, gets to threshold sets of atoms, and event sets to sets of atoms. A compact way to represent a change to the lattice is `deltas` allowing to easily and efficiently communicate such changes between *puts* and *gets*/handlers. For a set, a delta is an element; for a map, a key/value pair.

idempotence, meaning that `dtd = d` for any element `d`. In `LVish` terms, repeated puts or freezes have no effect, if the scheduler is allowed to occasionally duplicate work, it is possible to substantially save on synchronization costs. Since `LVish` computations are guaranteed to be idempotent, we could use such a scheduler (for now we use the standard Chase-Lev deque [10]). But idempotence also helps us deal with races between put and get/addHandler, Representation: data `LVars`, data status, put, remove, listener, etc Internally `par` monad represents computations in continuation passing style, in terms of their interpretation in the `IO` monad. Signature and type of the following methods along with short explanation: `GetLV` `putLV` `freezeLV` `Handler` pools and quiescence

An example to illustrate the use of `Lvish` library is performing a parallel graph traversal given a starting node and finding all the nodes reachable from it. We first create a new `LVar` of a set type, called `seen`. Then we attach an event handler to the

seen set. The `newHandler` function, defined elsewhere, takes 2 arguments, first is the LVar, `seen`, and second is the callback that we want to run every time an event arrives that is, every time a new node is added to the set. The callback takes that newly arrived node `node` as its argument, looks up nodes neighbors in the graph by calling a function `neighbors` (also defined elsewhere), and then maps the `insert` function over the list of neighbor nodes that `neighbors` returns. Then, we add the starting node to the `seen` set with `insert startNode seen` and the event handler does the rest. LVar operations can only run inside `Par` computations which are executed with one of the `run` functions. `runParThenFreeze`, executes a `Par` computation and then returns a frozen LVar at the end of it, after an implicit global barrier. So, if we have a `Par` computation that side-effects an LVar and we run it with `runParThenFreeze`, then we don't have to manually freeze the LVar ourselves, which means we don't have to manually quiesce, either! This function returns a computation of type `Par Det`, for deterministic. The `Det` is an effect level, meaning that the computation can only perform deterministic operations in particular, it's not allowed to do a freeze, and if it tried to, then that would be a static type error. If we had called `quiesce` and `freeze` explicitly in our code within `traverse` function, then it would return a computation of type `Par QuasiDet`, for quasi-deterministic. `traverse` starts off the computation and then returns the `seen` LVar. The LVar may well still be growing at the time `traverse` returns but, since `runParThenFreeze` cannot return unless all threads launched inside the `Par` computation are done, there's no threat to determinism, and we'll always get the same set of nodes as our answer. More generally, the fact that it's possible to safely return an LVar while it is still being written to by a handler makes it possible to set up dataflow-style, pipeline-parallel networks of LVars, letting us combine task parallelism and pipeline parallelism in the same deterministic program.

4 Summary

The summary shortly repeats the core ideas and results from the previous text. If the reader has problems understanding the summary he knows that he should go back to the relevant sections. Thus, the last section should consist of:

- a summary,
- an evaluation of what was done, importance of this work,
- what is left, what still needs to be done,
- short outlook into the future.

Last but not least, we can explain anything missing yet in the evaluation done in this paper. This allows to refer to what readers can expect from authors in the future.