Appendix

Contents

A Proofs

Theorem 4.1 (Synthesis Soundness).
$\forall I, d.\ (\bot, I \triangleright d) \rightarrow (d \vDash I)$

Proof.
Immediate from Theorem A.1.

Theorem A.1.
$\forall I, d, d', T.\ \exists T'.$
$d, I \triangleright d' \ \wedge$
$\Gamma \vdash d : T$
$\rightarrow$
$d' \vDash I \ \wedge$
$\Gamma \vdash d' : T'$
$\wedge\ \forall I'.$
$d \vDash I'$
$\rightarrow$
$d' \vDash I'$

Proof.
We assume
(1) $d, I \triangleright d'$
(2) $\Gamma \vdash d : T$
We prove
$d' \vDash I$
$\Gamma \vdash d' : T'$
Also, assuming
(3) $d \vDash I'$
We prove
$d' \vDash I'$

Induction on the derivation of [1]:
Case rule S-ID:
We have
(4) $d' = d$
(5) $d' \vDash I$
From [2] and [4]
(6) $\Gamma \vdash d' : T'$
From [3] and [4]
(7) $d' \vDash I'$
The conclusion is [5], [6] and [7].

Case rule S-JoinL:
We have
(8) $d = d_1 \bowtie d_2$
(9) $d' = d_1' \bowtie d_2$
(10) $d_1, I \triangleright d_1'$
By inversion on [2], [8], we have
(11) $\Gamma \vdash d_1 : T_1$
(12) $\Gamma \vdash d_2 : T_2$
By induction hypothesis on [10], [11]
(13) $d_1' \vDash I$
(14) $\Gamma \vdash d_1' : T_1'$
By Lemma A.2 on [13], [12], we have
(15) $d_1' \bowtie d_2 \vDash I$
By rule T-Join on [14], and [12], we have
(16) $\Gamma \vdash d_1' \bowtie d_2 : T_1' \cup T_2$
We show that
(17) $d_1' \bowtie d_2 \vDash I'$
By Lemma A.8 on [3] and [8], we have two cases
Case
(18) $d_1 \vDash I'$
By the induction hypothesis on [18], we have
(19) $d_1' \vDash I'$
By Lemma A.2 on [19], [12], we have
(20) $d_1' \bowtie d_2 \vDash I'$
Case
(21) $d_2 \vDash I'$
By Lemma A.3 on [21], [14], we have
(22) $d_1' \bowtie d_2 \vDash I'$
The conclusion is [15], [16] and [17].

Case rule S-JoinR:
Similar to rule S-JoinL

Case rule S-Add-1:
We have
(23) $I = [A]$
(24) $d' = d \bowtie (A \mapsto \bot)$
By rule T-Unit, we have
(25) $\Gamma \vdash \bot : \mathsf{Unit}$
By rule T-Map on [25], we have
(26) $\Gamma \vdash A \mapsto \bot : A \rightarrow \mathsf{Unit}$
By rule T-Join on [2] and [26], we have
(27) $\Gamma \vdash d \bowtie (A \mapsto \bot) : T \cup (A \rightarrow \mathsf{Unit})$
By rule I-Map-Key, we have
(28) $A \rightarrow \mathsf{Unit} \vDash [A]$
By rule I-UniR on [28], we have
(29) $T \cup (A \rightarrow \mathsf{Unit}) \vDash [A]$
By rule I-D on [27] and [29], we have
(30) $d \bowtie (A \mapsto \bot) \vDash [A]$
From [30], [24] and [23], we have
(31) $d' \vDash I$
From [27] and [24], we have
(32) $\Gamma \vdash d' : T \cup (A \rightarrow \mathsf{Unit})$
By Lemma A.5 on [3], we have
(33) $d \bowtie (A \mapsto \bot) \vDash I'$
From [33] and [24], we have
(34) $d' \vDash I'$
The conclusion is [31], [32], and [34].

Case rule S-Ext-2:
We have
(35) $d = (A \mapsto \mathsf{Unit})$
(36) $I = A \rightarrow A'$

(37) $d' = (A \mapsto A')$
By rule T-ID, we have
(38) $\Gamma \vdash A' : A'$
By rule T-MAP on [38], we have
(39) $\Gamma \vdash A \mapsto A' : A \to A'$
By rule I-MAP-MAP, we have
(40) $A \to A' \vDash A \to A'$
By rule I-D on [39] and [40], we have
(41) $(A \mapsto A') \vDash A \to A'$
From [41], [37] and [36], we have
(42) $d' \vDash I$
By Lemma A.6 on [3], [35] and [37], we have
(43) $d' \vDash I'$
The conclusion is [42], [39], and [43].

Case rule S-ADD-2:
Similar to case rule S-ADD-1.
Using rule T-ID instead of rule T-UNIT.
Using rule I-MAP-MAP instead of rule I-MAP-KEY.

Case rule S-EXT-3:
Similar to case rule S-EXT-2.
Using rule T-MAP-KEY instead of rule T-ID.
Using rule I-MAP-KEY and then rule I-MAP-MAP′ instead of rule I-MAP-MAP.

Case rule S-ADD-3:
Similar to case rule S-ADD-1.
Using rule T-MAP-KEY instead of rule T-UNIT.
Using rule I-MAP-KEY and then rule I-MAP-MAP′ instead of rule I-MAP-KEY.
Using Lemma A.9 instead of Lemma A.6.

Case rule S-UNI:
We have
(44) $I = I_1 \cup I_2$
(45) $d, I_1 \triangleright d''$
(46) $d'', I_2 \triangleright d'$
By induction hypothesis on [45], [2] and [3], we have
(47) $d'' \vDash I_1$
(48) $\Gamma \vdash d'' : T''$
(49) $d'' \vDash I'$
By induction hypothesis on [46], [47], [49] and [48], we have
(50) $d' \vDash I_2$
(51) $d' \vDash I'$
(52) $d' \vDash I_1$
(53) $\Gamma \vdash d' : T'$
By rule I-UNI on [52] and [50], we have
(54) $d' \vDash I_1 \cup I_2$
From [54] and [44], we have
(55) $d' \vDash I$
The conclusion is [55], [53], and [51].

Lemma A.2.
$\forall d, d', I, \Gamma, T.$
$d \vDash I \ \wedge$
$\Gamma \vdash d' : T \to$
$d \bowtie d' \vDash I$

Proof.
By inversion of rule I-D, rule T-JOIN, rule I-UNIL and rule I-D.

Lemma A.3.
$\forall d, d', I, \Gamma, T.$
$d' \vDash I \ \wedge$
$\Gamma \vdash d : T \to$
$d \bowtie d' \vDash I$

Proof.
Symmetric to Lemma A.2.

Lemma A.4.
$\forall \Gamma, d.\ \exists T.$
$\mathit{free}(d) \subseteq \mathit{dom}(\Gamma)$
$\to$
$\Gamma \vdash d : T$

Proof.
Immediate from structural induction on d.

Lemma A.5.
$\forall d, d', I.$
$d \vDash I \ \wedge$
$\mathit{free}(d') = \emptyset$
$\to$
$d \bowtie d' \vDash I$

Proof.
By Lemma A.2 and Lemma A.4.

Lemma A.6.
$\forall A, A', I.$
$(A \mapsto \bot) \vDash I$
$\to$
$(A \mapsto A') \vDash I$

Proof.
We assume
(1) $(A \mapsto \bot) \vDash I$
We prove
$(A \mapsto A') \vDash I$

By inversion on [1] (with rule I-D),
(2) $\emptyset \vdash A \mapsto \bot : T_2$
(3) $T_2 \vDash I$
By inversion on [2] (with rule T-Map),
(4) $T_2 = A \rightarrow T'$
(5) $\emptyset \vdash \bot : T'$
By inversion on [5] (with rule T-Unit),
(6) $T' = \mathsf{Unit}$
From [4] and [6]
(7) $T_2 = A \rightarrow \mathsf{Unit}$
From [3] and [7]
(8) $(A \rightarrow \mathsf{Unit}) \vDash I$
By inversion on [8] (with rule I-Map-Key),
(9) $I = [A]$
By rule I-Map-Key, we have
(10) $A \rightarrow A' \vDash [A]$
From [10] and [9]
(11) $A \rightarrow A' \vDash I$
By rule T-ID, we have
(12) $\Gamma \vdash A' : A'$
By rule T-Map on [12], we have
(13) $\Gamma \vdash A \mapsto A' : A \rightarrow A'$
By rule I-D on [13] and [11], we have
(14) $(A \mapsto A') \vDash I$

Lemma A.7.
$\forall T_1, T_2, I.$
$T_1 \cup T_2 \vDash I$
$\rightarrow$
$(T_1 \vDash I) \vee (T_2 \vDash I)$

Proof.
By induction on the derivation of $T_1 \cup T_2 \vDash I$.
The only inductive case is rule I-Uni.
The cases for the rule I-UniL, rule I-UniR are straightforward.

Lemma A.8.
$\forall d_1, d_2, I.$
$d_1 \bowtie d_2 \vDash I$
$\rightarrow$
$(d_1 \vDash I) \vee (d_2 \vDash I)$

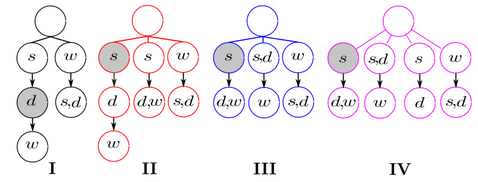
Proof.
By inversion on $d_1 \bowtie d_2 \vDash I$ and rule I-D, we have
(1) $\emptyset \vdash d_1 \bowtie d_2 : T$
(2) $T \vDash I$
By inversion on [1], we have
$T = T_1 \cup T_2$
$\emptyset \vdash d_1 = T_1$
$\emptyset \vdash d_2 = T_2$
Then, the conclusion is straightforward from Lemma A.7, and rule I-D.

Lemma A.9.
$\forall d, A, A', I.$
$d \bowtie (A \mapsto \bot) \vDash I$
$\rightarrow$
$d \bowtie (A \mapsto (A' \mapsto \bot)) \vDash I$

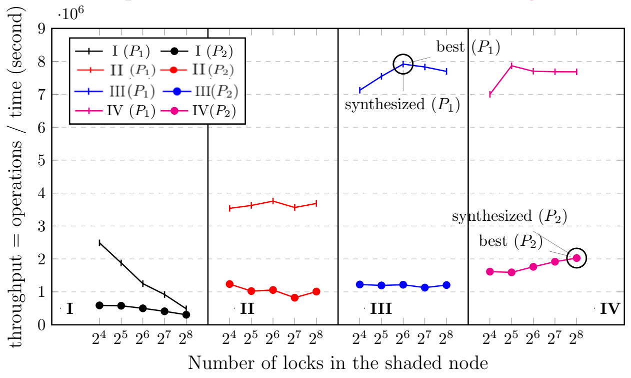
Proof.
Similar to Lemma A.6.

B Benchmarks

B.1 Graph Benchmark



(a) Decomposition candidates for the interface of Figure 4.(b)



(b) Benchmark: Graph. Workload: 90% Query, 10% Update.

Figure 11. Actual throughput for candidate representations and the synthesized representation. The optimum and synthesized candidates for both platforms P_1 and P_2 are the same.

Figure 11 shows the graph for the query-dominated workload with 1% updates that is elided from Figure 10 of the main body. In this experiment, we increase the size of the lock array for the nodes that are shaded in Figure 10.(a) from 16 to 256. The size of the lock array for the other nodes is constant, 128 in Figure 11.(b). As noted in the paper, we see a trend of throughput increase for the decompositions from I to IV. This suggest that wider decompositions deliver higher performance for query-dominated workloads.

B.2 Process Scheduler

This subsection presents the Process Scheduler benchmark adopted from [21]. The relation represents processes. The set of attributes are the process identifier *pid*, its namespace *ns*, its state *state* (that is running or idle) and its assigned processor *cpu*. The functional dependency $ns, pid \rightarrow state, cpu$ states that given a namespace and a process identifier, there are unique values for the the state and the processor.

We study the effect of increasing the number of locks on the throughput with a query-dominated workload with 1% updates in Figure 14 and also a workload with 10% updates in Figure 15. The query ratio is evenly distributed between the get methods. We increase the size of the lock array for the nodes that are shaded in Figure 13 from 128 to 1024. The size of the lock array for the other nodes is 128 for all the other nodes. Decompositions with shared leaf nodes are generally more efficient for update dominant scenarios because an update is done for all branches. However, the same sharing causes query dominated scenarios to have lower performance.

$$\begin{aligned}
\langle \mathcal{A} &:= \{pid, ns, state, cpu\}, \\
\mathcal{F} &:= \{ns, pid \rightarrow state, cpu\}, \\
\mathcal{I} &:= \{\langle ns \rightarrow [pid],\ 45\%\rangle, \\
&\quad \langle \langle ns, pid\rangle \rightarrow cpu,\ 45\%\rangle, \\
&\quad \langle state \rightarrow [cpu],\ 5\%\rangle\}, \\
\mathcal{P} &:= 5\%\rangle
\end{aligned}$$

Figure 12. Relational Specification of the Process Scheduler Use-case

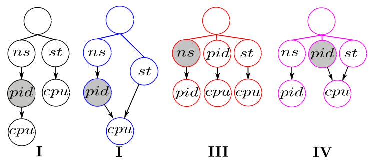


Figure 13. Decomposition candidates for the interface of Figure 12

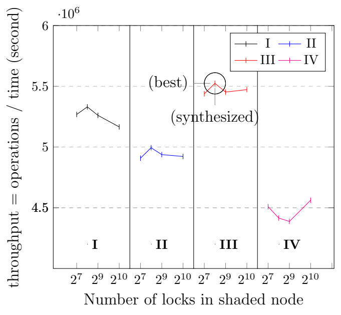


Figure 14. Benchmark: Process scheduler. Workload: 99% Query 1% Update

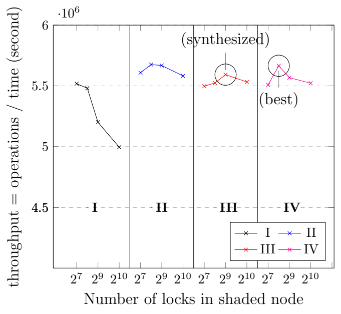


Figure 15. Benchmark: Process scheduler. Workload: 90% Query, 10% Update

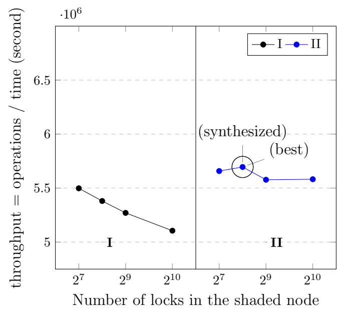


Figure 18

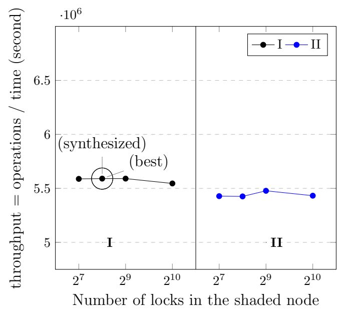


Figure 19

B.3 File System

This subsection presents the File System benchmark adopted from [21]. The set of attributes $\mathcal{A}$ is $\{parent, name, child\}$. Each *parent* directory entry has zero or more *child* directory entries, each with a distinct file *name*. The set of functional dependency $\mathcal{F}$ is the single dependency $parent, name \rightarrow child$. The interface $\mathcal{I}$ is the set of three access methods. The first one gets all the names in a directory in case of unmounting a filesystem. The second one gets a file given its parent directory and name. The third one gets the paths of a file in case of search.

We study the effect of increasing the number of locks on the throughput with a query-dominated workload with 1% updates in Figure 18 and also a workload with 10% updates in Figure 19. The query ratio is evenly distributed between the get methods. We increase the size of the lock array for the nodes that are shaded in Figure 17 from 128 to 1024. The size of the lock array for the other nodes is 128 for all the other nodes. As observed in the graph benchmark, increasing the size of the lock array of the inner map increases the cost of iteration and reduces performance.

$$
\begin{array}{rl}
\langle \mathcal{A} := & \{parent, name, child\}, \\
\mathcal{F} := & \{parent, name \rightarrow child\}, \\
\mathcal{I} := & \{\langle parent \rightarrow [name],\ 40\%\rangle, \\
& \langle\langle parent, name\rangle \rightarrow child,\ 40\%\rangle, \\
& \langle child \rightarrow [\langle parent, name\rangle],\ 10\%\rangle, \}, \\
\mathcal{P} := & 5\%\rangle
\end{array}
$$

Figure 16. Relational Specification of the File System Use-case

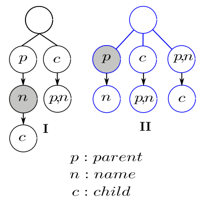


Figure 17. Decomposition for the File System Use-case

B.4 Sat4J

Queries can result a set of values and the user can apply the aggregation functions to results. However for convenience, we support aggregation queries and use them in this benchmark. Aggregation queries accept a predicate and an aggregate function on the output attributes. For example, the interface $A_1 \rightarrow [A_2], p, f$, given a value of the aggregation function f on all the corresponding values of the attribute A_2 that satisfy the predicate p.

This subsection presents the Sat4J benchmark adopted from [31]. The set of attributes $\mathcal{A}$ is $\{var, level, reason, posWatch\}$. The set of functional dependency $\mathcal{F}$ is the single dependency $var \rightarrow level, reason, posWatch$. The interface $\mathcal{I}$ is the set of three access methods.

The first, second and third get methods return the level, the reason and the posWatch values associated with the var respectively. The method $\langle[var], (\lambda var.\ T), count\rangle$ calculates the size of the map by counting all the *var*s in the data structure. The predicate on this interface returns true for all vars. The method $var?$ returns whether the relation contains var.

Figure 22 shows performance of two different workloads for this benchmark. Changing the number of locks has different effects on the throughput for the two workloads. The method $\langle[var], (\lambda var.\ T), count\rangle$ translates to iteration. When the method has a high ratio (50%), iteration impacts the throughput. If the ratio of the method is moderate (10%), increasing the number of locks reduces contention for other methods and improves the throughput but only up to a certain optimum.

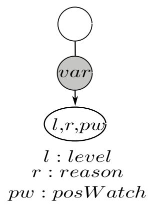


Figure 20. Decomposition for the Sat4J Use-case

$$\begin{aligned}
\langle \mathcal{A} &:= \{key, state, diskSize, inUse\},\\
\mathcal{F} &:= \{key \to state, diskSize, inUse\},\\
\mathcal{I} &:= \{\langle key \to \langle state, diskSize, inUse\rangle,\ 10\%\rangle,\\
&\quad \langle state \to [diskSize], (\lambda\ diskSize.\ T),\ sum,\ 40\%\rangle,\\
&\quad \langle state \to [\langle state, diskSize, inUse\rangle],\ 29\%\rangle,\\
&\quad \langle\langle state, inUse\rangle \to [\langle state, diskSize, inUse\rangle],\ 20\%\rangle\},\\
\mathcal{P} &:= 1\%\rangle
\end{aligned}$$

Figure 23. Relational Specification of the Ztopo

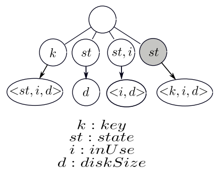


Figure 24. Decomposition for the Ztopo Use-case

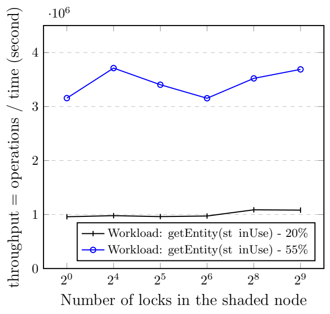


Figure 25. Performance for the ZTopo Use-case.

$$\begin{aligned}
\langle \mathcal{A} &:= \{var, level, reason, posWatch\},\\
\mathcal{F} &:= \{var \to level, reason, posWatch\},\\
\mathcal{I} &:= \{\langle var \to level,\ 22\%\rangle,\\
&\quad \langle var \to reason,\ 22\%\rangle,\\
&\quad \langle var \to posWatch,\ 1\%\rangle,\\
&\quad \langle var?,\ 5\%\rangle,\\
&\quad \langle [var], (\lambda\ var.\ T), count,\ 50\%\rangle\},\\
\mathcal{P} &:= 1\%\rangle
\end{aligned}$$

Figure 21. Relational Specification of the Sat4J Use-case

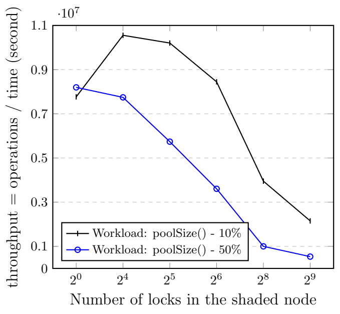


Figure 22. Performance for the Sat4J Use-case. The method $poolsize()$ is $\langle [var], (\lambda var.\ T), count\rangle$.

B.5 ZTopo

This subsection presents the ZTopo adopted from [31]. The set of attributes $\mathcal{A}$ is $\{key, state, diskSize, inUse\}$. The set of functional dependency $\mathcal{F}$ is the single dependency $key \to state, diskSize, inUse$. The interface $\mathcal{I}$ is the set of five access methods. The interface results in the decomposition shown in Figure 24.

In Figure 25, we study the effect of increasing the number of locks for *state* on the throughput with two workloads. We increase the size of the lock array for *state* from 128 to 1024. The size of the lock array for the other nodes is the constant, 128.

The two workloads have different ratios for the two methods $getEntity()$ that is $\langle state, inUse\rangle \to [\langle state, diskSize, inUse\rangle]$ and $getTotalDiskSize()$ that is $\langle state \to [diskSize], (\lambda\ diskSize.\ T),\ sum$. In the first workload, the ratios of the two methods are 20% and 40% respectively. In the second workload, the ratios of the two methods are 55% and 5% respectively. The method $getTotalDiskSize()$ aggregates list elements; thus, the throughput decrease as its ratio is increased.

C Related Work

Synthesizing data structures. The importance of data structure synthesis has been recognized since 70s. Iterator inversion [15] automatically constructs iterators over data representations. SETL [41] dynamically maps abstract set and map types to concrete implementations. PReps [13] inputs relational specifications and annotations that aid compilation to efficient data structures. DiSTiL [44] presents a declarative language for data structure specifications and replaces specifications with low-level implementations. In contrast, Leqsy supports more general relational specifications, learns the performance model and statically synthesizes more elaborate concurrent data structures.

RelC [21, 22] synthesizes both sequential and concurrent data representations. Cozy [31] synthesizes efficient sequential data structures using a static cost model. In a follow-up work [30], it supports operations across data structures and automatically synthesizes updates as well as queries. In contrast, Leqsy learns a performance model instead of using a static model and synthesizes concurrent data structures.

Sketching [45, 46] automatically completes a sketch implementation of a concurrent data structure. Boosting [23] and Semantic Locking [19] use commutative pessimistic synchronization while Predication [6] and Transactional Libraries [47] use optimistic synchronization [24] to implement composable concurrent data structures. Users can use these techniques to compose atomic structures manually. However, Leqsy automatically synthesizes them.

Learning and concurrent data structures. Machine learning has aided concurrent systems. It has been used to predict the number of concurrent threads for optimum performance. [38, 39, 48]. Smartlocks [17] uses machine learning models to adapt spin-locks to dynamic workloads. Smart Data Structures [16] use online learning to adapt the data structure to varying workloads. Similarly, Chameleon [43] dynamically profiles client calls on data structures to suggest efficient data structure implementations. In contrast, Leqsy learns a performance model and statically synthesizes the most efficient data structure for a target workload. Data Calculator [27] synthesizes data structures and predicts their performance based on user-defined layout specifications and target architectures. It learns and uses performance models for its primitives. In contrast, Leqsy does not require layouts, supports put and query operations, learns the performance model of layouts, and explores the space of layouts.

Synthesizing concurrent programs. Various projects [5, 12, 14, 20, 26, 33, 55] infer atomic sections and insert locks to implement them. Others [2, 8, 9, 18] apply semantic-preserving transformations to efficiently synchronize, repair or verify synchronization. Similarly, Paraglider [51] and AGS [52] explore the execution space to iteratively add synchronization that removes violating interleavings. Other works [1, 3, 28] automatically insert fence instructions to data structures. RCU [34] and RLU [32] automatically reduces latency of read/write accesses to the data structure. Elixir [36, 37] synthesizes synchronization for parallel graph processing. These works synthesize synchronization with no quantitative [4, 7, 10, 11] measures. Quantitative synthesis [50] however, uses analytical performance models to synthesize concurrent programs. On the other hand, Leqsy learns and uses the performance model for quantitative comparisons.