Low-level strongly typed dataframes for machine learning and statistical computing in resource-constrained devices

Jayaraj Poroor
Paanini Inc.
jayaraj.poroor@paanini.com

***Abstract*— Strongly typed dataframe-like abstraction suitable for resource-constrained devices is presented. The proposed dataframe type system assigns types to memory addresses rather than symbolic names, supports memory reuse, and has zero runtime overheads. A subsumption relation is defined on dataframe types that allows flexible, strongly-typed access to rows, columns, and fragments. A specialized form of separation logic is proposed to formally reason about dataframe typing, allowing compositional reasoning. Hoare-style formal rules are defined to reason about the type-safety of programs operating on dataframes.**

I. Introduction

Dataframes [1] are important data structures for in-memory statistical computing and machine learning, found in nearly all major platforms such as R[1], Python (Pandas)[2], and Apache Spark [3]. Recently, there has been tremendous interest in on-board data analysis and machine learning in resource constrained devices [4], [5], [6]. However, a flexible dataframe-like abstraction is absent in such systems. In this paper we propose low-level and flexible dataframe-like data structures for constrained environments.

Our dataframes offer strong static type safety guarantees, assigns types to memory addresses rather than abstract symbolic names, and have zero runtime bookkeeping overheads - important considerations in constrained devices. Our type system allows flexible, strongly typed access to row ranges, columns, or row and column fragments. To visually illustrate, consider storing a time series array of samples from a temperature sensor (16-bit integer), a switch input (1 byte), and a 1-axis accelerometer (32-bit integer) as follows:

100	t_0	s_0	a_0
	t_1	s_1	a_1
	..	..	..
	t_m	s_m	a_m

By using the address *100* we may wish to access either: (a) the entire vector of tuples of type $(i16 \times i8 \times i32)[m]$; (b) the row vector $[t_0, s_0, a_0]$ of type $i16 \times i8 \times i32$, or, (c) the column vector $[t_0, t_1, ..., t_m]$ of type $i16[m]$. iN represents N-bit integer type. Our type system enables inferring such fragment types from the dataframe type and also inferring the dataframe type from its fragment types. The main technical contributions of this paper are:

$$
\begin{array}{lll}
\iota & ::= & \\
& | & alloc[\tau]\ a \\
& | & free[\tau]\ a \\
& | & zero[w]\ a, s, c \\
& | & transform_T[\tau_a, \tau_b]\ a, s_a, b, s_b, c \\
& | & transform_T[\tau]\ a, s, c
\end{array}
\tag{II-.1}
$$

Fig. 1. Instruction set of our abstract VM.

- We define a *dataframe type constructor* and a subsumption relation on the dataframe types that permits strongly typed access to dataframe fragments such as individual columns or a range of rows.
- We propose a specialized form of separation logic [7] to reason about memory type assignments. This allows us to: (a) reason independently about dataframe fragments (*local reasoning*) and then compose them in a simple manner; (b) reason in presence of memory re-use.
- We develop a formal model for our separation logic using an *abstract heap* holding type information instead of a concrete *value heap.*
- We define formal rules for verifying the memory safety and type safety of the proposed dataframes.

II. The Abstract Virtual Machine

We present the dataframe typing in the context of an abstract virtual machine (VM) with a small, generic vector instruction set that operate on dataframe fragments in memory. The *alloc* and *free* operations are for allocating and freeing dataframes and are statically interpreted by our type system (the addresses for allocation/free must be present in the assembly program). We may define a *realloc* operator to reallocate dataframes, using *free* and *alloc*: $realloc[\tau', \tau]\ a \triangleq free[\tau]\ a; alloc[\tau']\ a.$

The *zero* instruction is for vector initialization. The *transform$_T$* is a class of transformation operations, indexed by the transformation function T that operate on dataframes with elements of type τ_a with resultant elements of type τ_b. We also define a variant for in-place transformations. We have a special *byte* type representing unallocated memory.

III. Memory typing

A dataframe type is specified using a parameterized type term of the form: $\tau\{s\}[n]$, where: (a) τ is the element

978-1-7281-4177-0/19/$31.00 © 2019 IEEE

type; (b) s is the number of memory locations between each element, which we call *stride length*, and, (c) n is the total count of elements. The type τ may be a scalar type, a product type, or another dataframe type. The type term must satisfy the invariant:

$$\tau\{s\}[n] \implies size(\tau) \leqslant s \land n > 0 \tag{III-.1}$$

The typing assertion $@a : \tau$ states that starting at the address a a data item of type τ exists. The predicate $@a : \tau\{s\}[n]$ represents a dataframe structure starting at memory location a, having element type τ, stride s, and total of n elements. We use the symbol α to denote typing assertions of the form $@a : \tau$.

We define a *size* function that maps types to natural numbers, representing the number of memory locations need to hold a value of the type. For aggregate types, the size function can be inductively defined in terms of constituent types:

$$\begin{aligned} size(\tau_a \times \tau_b) &= size(\tau_a) + size(\tau_b) \\ size(\tau_a + \tau_b) &= max(size(\tau_a), size(\tau_b)) \\ size(\tau\{s\}[n]) &= s \times n \end{aligned} \tag{III-.2}$$

We identify a scalar type with a vector of count 1. When *count* = 1, we may omit the *count* in the type term. The following type terms are equivalent:

$$\tau \equiv \tau\{size(\tau)\}[1] \equiv \tau\{size(\tau)\} \tag{III-.3}$$

Product and vector types are equivalent when element types are same:

$$\tau\{size(\tau)\}[n] \equiv \tau \times \tau\{size(\tau)\}[n-1] \ where \ n > 1 \tag{III-.4}$$

As an example, consider the scenario of acquiring per-second samples from a 1-phase current sensor for 1 minute, applying conversion formula to get ampere-values, and then checking for threshold crossing for each sample. This can be expressed as the following typed assembly program (the dataframe is allocated at address *100*):

alloc	$[(u32 \times float \times bool)\{9\}[60]]$ 100
convert	$[u32, float]$ 100, 9, 104, 9, 60
pointwise_gt	$[float, bool]$ 104, 9, 108, 9, 60

A. *The dataframe memory typing and subsumption*

We define rules for inferring larger dataframe structures from the knowledge of their fragments as follows:

$$\frac{@a : \tau_a\{s\}[n] \quad @b : \tau_b\{s\}[n]}{@a : (\tau_a \times \tau_b)\{s\}[n]} \ a + size(\tau_a) = b \tag{III-A.1}$$

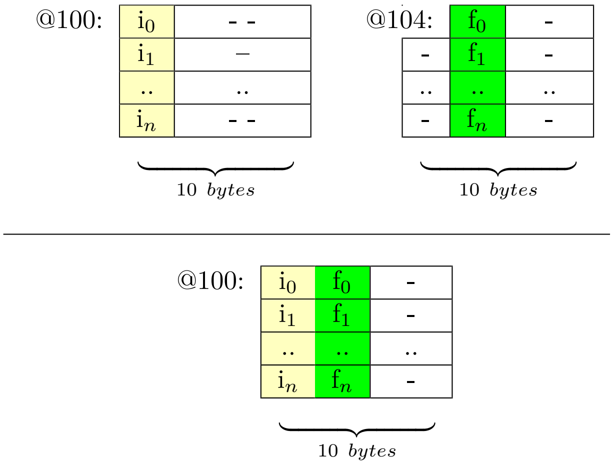
$$\frac{@a : \tau_a \quad @b : \tau_b}{@a : \tau_a \times \tau_b} \ a + sizeof(\tau_a) = b \tag{III-A.2}$$

$$\frac{@a : \tau\{s\}[m] \quad @b : \tau\{s\}[n]}{@a : \tau\{s\}[m+n]} \ a + s \times m = b \tag{III-A.3}$$

$$\frac{@a : \tau\{s_n\}[n] \quad @a : \tau\{s_m\}[m]}{@a : \tau\{g\}[max(\frac{n \times s_n}{g}, \frac{m \times s_m}{g})]} \ g = \gcd(s_n, s_m) \tag{III-A.4}$$

Consider the rule III-A.1. We illustrate the intuition behind this rule with a diagram (below), which can be read as:

- If we know that at address *100* there is a vector of 4-byte integer values $[i_0, i_1, ...i_n]$ with $n+1$ elements and a stride of *10 bytes*, and,
- We know that at address *104* there is a float vector $[f_0, f_1, ..., f_n]$ also with $n+1$ elements and stride of *10 bytes*, then,
- We may infer that at address *100* there is a vector of integer × float tuples $[(i_0, f_0), (i_1, f_1), ..., (i_n, f_n)]$ with $n+1$ elements and stride of *10 bytes.*



The rule III-A.4 is useful for accessing a timeseries vector in a downsampled fashion.

The above inference rules induce a partially ordered subsumption relation $\sqsubseteq$ on dataframe structure assertions. If $@a : \tau_b$ and $@b : \tau_b$ appear in the premise and $@c : \tau_c$ in the conclusion of any of the above rules then the following holds:

- $@a : \tau_a \sqsubseteq @c : \tau_c$ and $@b : \tau_b \sqsubseteq @c : \tau_c$. This means that the heap representing $@a : \tau_a$ ($@b : \tau_b$) is a subset of the heap representing $@c : \tau_c$.

The relation $x \sqsubseteq y$ may be read as: x *subsumes* y, x is a *substructure of* y, or y is a *superstructure of* x.

We may indirectly refer to dataframe structures by storing the dataframe structure parameters, viz., address, stride, and count. We may make such references without specifying the exact element type. We use syntactic equivalence: $@a : ?\{s\}[n] \triangleq \exists \tau \cdot @a : \tau\{s\}[n]$

We use β to range over typing assertions in which the element type is not specified, i.e., β has the form $@A : ?\{S\}[C]$ and β_y denotes the corresponding typing assertion obtained by filling in the element type as y, i.e. given $\beta = @A :?\{S\}[C]$,

$$\begin{array}{lcl} \beta_{?t} & \triangleq & @A :?t\{S\}[C] \\ \beta_{?t|w|} & \triangleq & @A :?t|w|\{S\}[C] \\ \beta_{\tau} & \triangleq & @A : \tau\{S\}[C] \end{array} \tag{III-A.5}$$

We use α to range over concrete typing assertions.

We define an abstract least upper bound operator $\sqcup_{?\tau}$ on abstract dataframe structure references, wherein the exact element type is omitted. .

$$\begin{array}{rcl} @a :?\{s\}[m] \sqcup_{?\tau} @b :?\{s\}[n] & = & @c :?\{s\}[p] \\ & & \textit{if } a \leqslant b < a + s \\ \multicolumn{3}{r}{\textit{where } c = min(a, b),\ p = max(m, n)} \\ \\ @a :?\{s\}[m] \sqcup_{?\tau} @b :?\{s'\}[n] & = & @a :?\{g\}[p] \\ \multicolumn{3}{r}{\textit{if } a < b \leqslant a + s \times m \ \textit{ and } b \geqslant a + s} \\ \multicolumn{3}{r}{\textit{where } g = gcd(s, s'), p = \frac{b-a+1}{g}} \end{array} \tag{III-A.6}$$

A partial order $\sqsubseteq_{?\tau}$ on abstract typing assertions is defined by the $\sqcup_{?\tau}$:

$$\alpha \sqsubseteq_{?\tau} \alpha \sqcup_{?\tau} \alpha' \tag{III-A.7}$$

IV. A separation logic of dataframe typing assertions

A. Syntax

We propose a variant of the standard separation logic to specify and verify the type safety of memory operations. Separation logic gives us the advantage of employing local and in-place reasoning [8], which simplify the specifications and the proofs. The separating conjunction allows us to independently specify typing assertions about dataframe fragments and then compose them. Moreover, separation logic simplifies reasoning in the presence of concurrency [9]. Hence it will also allow us easily extend our reasoning to cover the limited forms of concurrency found in constrained devices.

Our separation logic syntax is defined below:

$$\begin{array}{lcl} L & ::= & @a :?\{s\}[n] \\ & | & L_0, L_1 \ \textit{where } L_0, L1 = L_1, L_0 \\ \\ P, Q & ::= & \textit{emp} \\ & | & @a : \tau \\ & | & P * Q \\ \\ \textit{where } \tau & ::= & \tau'\{s\}[n] \\ & | & \textit{addr}(L) \\ & | & \textit{stride}(L) \\ & | & \textit{count}(L) \\ & | & \textit{val}(M) \ \textit{where } M : \tau_i \end{array} \tag{IV-A.1}$$

We briefly explain the syntax below:

- The symbol L denotes a commutative list of abstract dataframe typing assertions.
- $@a : \tau$ asserts that a data structure of type τ exists, starting at address a.
- The type τ in the assertion may be a dataframe type or an integral refinement type as elaborated below.
- The dataframe structures can be referred to indirectly using parameters stored in memory locations. We may wish to abstract out the specific values in such memory locations and simply maintain the information about the largest dataframe structure being referred to by these parameters. This is useful when, due on conditional execution, different values are filled in the locations referring to a dataframe structure. Instead of tracking all these concrete values, we just need to track an abstract reference to the largest dataframe structure. The types $\textit{addr}(L), \textit{stride}(L), \textit{count}(L)$ are refinement types [10] of a predefined integral type τ_i that allows us to do so. Consider the assertion $@a : \textit{addr}(L)$. This asserts that:
  - the memory location at address a contains an integer value A, such that:
  - for each dataframe structure specified in L, A is the address of one its substructures.
- For our discussion we use *ref* to refer to either of *addr*, *stride*, or *count*, i.e., $ref \in \{\textit{addr}, \textit{stride}, \textit{count}\}$.
- $\textit{val}(M)$ is a singleton type as in the TALx86 typed machine language [11], that allows us to track specific values in memory cells.
- We assume that τ_i has enough word width to hold memory addresses.
- $*$ is the standard separating conjunction.

B. Semantics

Our model differs from the standard separation logic models in the literature in two ways:

- Since at machine level there are no named variables, our separation logic model does not have a *s*tore that maps variables to integers.
- Our heap is abstract, holding only the type information. To distinguish from the concrete heap (usually denoted by the symbol h), we use the symbol h_t to refer to our abstract heap.

We define a heap as a finite function from natural numbers, representing memory addresses, to a lifted set of basic types:

$$\text{Heaps} \triangleq \mathbb{N} \rightharpoonup_{fin} \text{BasicNames} \cup \{\bot\} \cup \textit{values}(\tau_i) \tag{IV-B.1}$$

A summary of the notations used (wherever applicable, our notations are consistent with the existing literature on separation logic [8]):

- We use the variable name τ_p to range over primitive types such as N-bit integers (iN). The *byte* type represents the type of unallocated byte-sized memory locations.

- We use the term basic type to refer to primitive types along with the *byte* type. The set *BasicNames* refers to the set of symbolic names corresponding to basic types. Given a basic type τ_b we use the notation $nm(\tau_b)$ to refer to the symbolic name corresponding to the type.
- $\perp$ represents an illegal type which can be assigned to locations which must not be accessed.
- The set $\mathit{values}(\tau_i)$ represent the set of values that a location of type τ_i can take. Type assertions involving the singleton and other refinement types are interpreted using the actual value the memory location holds.
- h_t represents an abstract heap holding type information, i.e., $h_t \in$ Heaps.
- $dom(h_t)$ denotes the the domain of the abstract heap h_t.
- $h_{t1} \# h_{t2}$ denotes that $dom(h_{t1}) \cap dom(h_{t2}) = \varnothing$.
- $h_{t1} \uplus h_{t2}$ is union heap functions with disjoint domains (undefined if domains are not disjoint).

The ? or $?X$ in our typing assertions refers to existentially quantified type variables. The satisfaction relation $h_t \vDash P$, which states that the assertion P holds for the abstract heap h_t is defined as follows:

$$\begin{array}{lll}
h_t \vDash @a : \tau\{s\}[n] & \text{if} & \forall x \in 0..n-1 \cdot \\
 & & \qquad h_t \vDash @(a + s{\times}x) : \tau \\
h_t \vDash @a : \tau_1 \times \tau_2 & \text{if} & h_t \vDash @a : \tau_1 \text{ and} \\
 & & h_t \vDash @(a + size(\tau_1)) : \tau_2 \\
h_t \vDash @a : \tau_b & \text{if} & h_t(a) = nm(\tau_b) \text{ and} \\
 & & \forall x \in 1..size(\tau_b) \cdot h_t(a+x) = \perp \\
h_t \vDash @a : val(M) & \text{if} & h_t(a) = M \text{ and} \\
 & & \forall x \in 1..size(\tau_i) \cdot h_t(a+x) = \perp \\
h_t \vDash @a : addr(L) & \text{if} & h_t(a) = A \ s.t \\
 & & \forall \alpha \in L \cdot @A :?\{?S\}[?N] \sqsubseteq_{?\tau} \alpha \\
 & & \mathit{and} \\
 & & \forall x \in 1..size(\tau_i) \cdot h_t(a+x) = \perp \\
h_t \vDash @a : stride(L) & \text{if} & h_t(a) = S \ s.t \\
 & & \forall \beta \in L \cdot @?A :?\{S\}[?N] \sqsubseteq_{?\tau} \beta \\
 & & \mathit{and} \\
 & & \forall x \in 1..size(\tau_i) \cdot h_t(a+x) = \perp \\
h_t \vDash @a : count(L) & \text{if} & h_t(a) = N \ s.t \\
 & & \forall \beta \in L \cdot @?A :?\{?S\}[N] \sqsubseteq_{?\tau} \beta \\
 & & \mathit{and} \\
 & & \forall x \in 1..size(\tau_i) \cdot h_t(a+x) = \perp \\
h_t \vDash P * Q & \text{if} & \exists h'_t, h''_t \cdot h'_t \# h''_t, h'_t \uplus h''_t = h_t, \\
 & & h'_t \vDash P,\ h''_t \vDash Q \text{ (note below)}^1
\end{array} \tag{IV-B.2}$$

C. Axioms and syntactic equivalence on typing assertions

The following condition allows memory typing assertions to be split and merged over the separating conjunction:

$$(a + \text{size}(\tau_a) = b) \wedge (@a : \tau \sqsubseteq @a : \tau_a) \wedge (@a : \tau \sqsubseteq @b : \tau_b) \tag{IV-C.1}$$

Given the above condition the following formula is valid:

$$(@a : \tau_a * @b : \tau_b) \iff @a : \tau \tag{IV-C.2}$$

Some vector instructions depend only on the word size of element types. For simplifying assertions about such instructions, we use the following syntactic equivalence:

$$@a : \tau|w|\{s\}[n] \triangleq @a : \tau\{s\}[n] \wedge size(\tau) = w \tag{IV-C.3}$$

If the specific concrete type is not relevant for the assertion, we use the syntax:

$$@a :?|w|\{s\}[n] \triangleq \exists \tau \cdot @a : \tau|w|\{s\}[n] \tag{IV-C.4}$$

The following rules can be used to generate refinement type assertions for address, stride, and count references from singleton type assertions.

$$\begin{array}{c}
@a : val(A) * @b : val(B) * @c : val(C) \implies \\
@a : addr(\beta) * @b : stride(\beta) * @c : count(\beta) \\
\mathit{where}\ \beta = @A :?\{B\}[C]
\end{array} \tag{IV-C.5}$$

In the inference rule, ref_j is either *addr*, *stride*, or *count*.

$$\frac{\begin{array}{c} P \implies @a_0 : ref_0(L_0) * ... * @a_k : ref_k(L_k) \\ P \implies @a_0 : ref_0(L'_0) * ... * @a_k : ref_k(L'_k) \end{array}}{P \implies @a_0 : ref_0(L_0 \cup L'_0) * ... * @a_k : ref_k(L_k \cup L'_k)}$$

Several instructions do not modify the type assignments and therefore have post-conditions same as the pre-conditions or may modify only some portions of the pre-condition. For syntactic simplicity we use the following syntactic equivalences to represent the post condition which is same as the pre-condition :

$$\begin{array}{lcl}
\{P\}C\{-\} & \triangleq & \{P\}C\{P\} \\
\{@a : \tau\}C\{@a : -\} & \triangleq & \{@a : \tau\}C\{@a : \tau\}
\end{array} \tag{IV-C.6}$$

V. Proving type and memory safety

We may use our typing assertions to prove type and memory safety of programs with dataframe operations using proof rules in the style of Hoare logic [12]. In Hoare logics, rules are specified as triples in the form of, *pre-condition, program schema, post-condition.* This may be read as: if the pre-condition holds for the memory state before the program execution and we execute the program then the post-condition will hold for the memory state after execution. In our case pre-condition and post-condition assertions on memory states are our typing assertions.

(1) The following rule states that if we have enough un-allocated memory at address a, and we perform *alloc* operation with dataframe type τ, then we get the dataframe of type τ at address a.

$$\begin{gathered}\{@a : \mathit{byte}[\mathit{size}(\tau)]\} \\ \mathit{alloc}[\tau]\ a \\ \{@a : \tau\}\end{gathered} \tag{V-.1}$$

(2) This rule states that if we have an typed dataframe at address a, then we may free the memory at address a to get back the un-allocated bytes at address a.

$$\begin{gathered}\{@a : \tau\} \\ \mathit{free}[\tau]\ a \\ \{@a : \mathit{byte}[\mathit{size}(\tau)]\}\end{gathered} \tag{V-.2}$$

It is easy to see that if we compose rules (1) and (2), i.e., if we allocate memory and free again, then we start and end with un-allocated sequence of bytes.

(3) This rule specifies that a *zero* operation with datatype width w may be executed on address a provided the target dataframe's element type has width w. Moreover, the operation does not alter the memory typing.

$$\begin{gathered}\{@a : \mathit{addr}(\beta) * @s : \mathit{stride}(\beta) * @c : \mathit{count}(\beta) * \beta_{?|w|}\} \\ \mathit{zero}[w]\ a, s, c \\ \{-\}\end{gathered} \tag{V-.3}$$

(4) This rule specifies that a transformation operation from element type τ_a to τ_b may be performed as long as the destination address b holds a dataframe fragment with element type whose width is same as the target type τ_b. The resultant memory at b has the element type τ_b. A variant of this could be defined for in-place transformations. The index function must be have a consistent type: $T : \tau_a[.] \to \tau_b[.]$.

$$\begin{gathered}\{@a : \mathit{addr}(\beta) * @s_a : \mathit{stride}(\beta) * @b : \mathit{addr}(\beta')* \\ @s_b : \mathit{stride}(\beta') * @c : \mathit{count}(\beta, \beta') * \beta_{\tau_a} * \beta'_{?|\mathit{size}(\tau_b)|}\} \\ \mathit{transform}_T[\tau_a, \tau_b]\ a, s_a, b, s_b, c \\ \{@a : - * @s_a : - * @b : - * @s_b : - * @c : - * \beta_{\tau_a} * \beta'_{\tau_b}\}\end{gathered} \tag{V-.4}$$

VI. Related work

Dataframe implementations in higher level languages [1], [2], [3], are not suitable for constrained environments owing to dynamic allocation requirements and data structure overheads. The existing work on typed assembly languages [11] have focused on type safety for basic memory instructions and not on high-level dataframe-like structures. TinyDB [13] is a small footprint query processing engine with SQL-like declarative query language for on-board conditional data acquisition, filtering and aggregation in constrained devices. However, unlike our dataframes, the SQL-like language of TinyDB is restricted to a small class of applications. Maté [14] is a small stack-based interpretive VM for wireless sensor network nodes (motes). It lacks higher level data processing capabilities with static type safety of our dataframes.

VII. Conclusion & future work

Our results show that a low level dataframe abstraction suitable for constrained environments can be developed with strong type and memory safety guarantees, which are important in such systems. Our dataframes could also be used for efficient and safe data processing within the web assembly [15] environment. As future work we intend to investigate higher level operations on the dataframes and implement a small footprint verifier for constrained devices.

References

[1] R Core Team et al. R: A language and environment for statistical computing. 2013.

[2] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[3] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[4] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3:637–646, 2016.

[5] Aditya Kusupati, Manish Singh, Kush Bhatia, Ashish Kumar, Prateek Jain, and Manik Varma. Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network. In *Advances in Neural Information Processing Systems*, pages 9031–9042, 2018.

[6] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. Protonn: compressed and accurate knn for resource-scarce devices. In *International Conference on Machine Learning*, pages 1331–1340, 2017.

[7] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[8] Peter W O'Hearn. A primer on separation logic (and automatic program verification and analysis)., 2012.

[9] Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, August 2016.

[10] Rowan Davies and Frank Pfenning. *Practical refinement-type checking*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2005.

[11] K Crary, Neal Glew, Dan Grossman, Richard Samuels, F Smith, D Walker, S Weirich, and S Zdancewic. Talx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*, pages 25–35, 1999.

[12] Richard Bornat. Proving pointer programs in hoare logic. In *International Conference on Mathematics of Program Construction*, pages 102–126. Springer, 2000.

[13] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[14] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. In *ACM Sigplan Notices*, volume 37, pages 85–95. ACM, 2002.

[15] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *ACM SIGPLAN Notices*, volume 52, pages 185–200. ACM, 2017.