Ownership Refinement Types for Pointer Arithmetic and Nested Arrays

Yusuke Fujiwara
Kyoto University, Japan

Yusuke Matsushita
Kyoto University, Japan

Kohei Suenaga
Kyoto University, Japan

Atsushi Igarashi
Kyoto University, Japan

Abstract

Tanaka et al. proposed a type system for verifying functional correctness properties of programs that use arrays and pointer arithmetic. Their system extends ConSORT– a type system combining fractional ownership and refinement types for imperative program verification – with support for pointer arithmetic. Their idea was to extend fractional ownership so that it can depend on an array index. Their formulation, however, does not handle nested arrays, which are essential for representing practical data structures such as matrices. We extend Tanaka et al.'s type system to support nested arrays by generalizing the notion of ownership to be able to refer to the indices of the outer arrays and prove the soundness of the extended type system. We have implemented a verifier based on the proposed type system and demonstrated that it can verify the correctness of programs that manipulate nested arrays, which were beyond the reach of Tanaka et al.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases aliasing, fractional ownership, program verification, refinement types, type systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.6

Related Version *Full Version*: `https://arxiv.org/abs/2604.22361` [15]

Supplementary Material *Software*: `https://zenodo.org/records/19521221`
Software (ECOOP 2026 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.12.1.23`

Funding This work is partially supported by JSPS KAKENHI Grant Number 20H05703, Japan.
Yusuke Matsushita: His research was supported in part also by the Hakubi Project at Kyoto University and JSPS KAKENHI Grant Number JP24KJ0133.
Kohei Suenaga: His research was supported in part also by JST CREST Grant Number JPMJCR2012, Japan, and JSPS KAKENHI Grant Number 25H01113, Japan.

Acknowledgements We appreciate the reviewers' constructive comments. We are also grateful to Naoki Kobayashi, Tsubasa Matsumoto, Ken Sakayori, and Izumi Tanaka for valuable comments and discussions on this work. Takashi Suwa helped us test the artifact. The second author is currently hosted by MPI-SWS as a JSPS Overseas Research Fellow.

1 Introduction

Type-based automated program verification has been a popular methodology for making software reliable. Among various type systems proposed so far, recent years have seen significant success of *refinement type systems* [26, 20] – type systems that can constrain

values using predicates called *refinement predicates* – to guarantee properties that cannot be expressed with base types alone. For example, a refinement type system can express the type $\{\nu : \mathbf{int} \mid \nu > 0\}$ of positive integers using the refinement predicate $\nu > 0$, which is more expressive than its simple type **int**. Refinement type systems have been applied to various types of programs including functional programs [20, 49], object-oriented programs [44], and smart contracts [31, 33].

However, applying refinement types to imperative programs with pointers and aliasing has been less popular than for other kinds of programs. A naive way of introducing refinement types for an imperative language would be to incorporate reference types. For example, such a type system would have type $\{\nu : \mathbf{int} \mid \nu > 0\}\,\mathbf{ref}$ for pointers to positive integers. The type system would be designed to be flow-sensitive: typing rules would be designed as if the type of each variable were updated by each statement. For example, if x has type $\{\nu : \mathbf{int} \mid \nu > 0\}\,\mathbf{ref}$ just before a statement $x := -1$ that updates the memory cell pointed to by x to -1, then the type of x just after this statement would be $\{\nu : \mathbf{int} \mid \nu = -1\}\,\mathbf{ref}$.

One major challenge in this approach is the *strong update* problem, explained below. Suppose pointers x and y both have type $\{\nu : \mathbf{int} \mid \nu > 0\}\,\mathbf{ref}$ just before a statement $x := 1$. Then, the type of x after this statement is updated to a type like $\{\nu : \mathbf{int} \mid \nu = 1\}\,\mathbf{ref}$. However, naively updating only the refinement type of x is not enough in general; if x and y are the same pointer, then the type of y must also be updated since the memory cell pointed to by y is updated by this statement.

To address this issue, Toman et al. [47] proposed a type system ConSORT. Their type system, instead of conducting static must-alias analysis, constrains aliases based on types. Concretely, their reference type is augmented with information called *(fractional) ownership*, with which their type system imposes the following invariants for each memory cell:

- there is at most one pointer that (1) can be used for updating and reading the memory cell and (2) is associated with a non-trivial (i.e., not $\top$) refinement predicate; and
- there may be additional pointers that (1) can be used only for reading from the memory cell and (2) are associated with a non-trivial refinement predicate.

With these constraints, the type system does not need to handle aliasing in a strong update, since it is guaranteed that there are no aliases reading non-trivial information from the updated memory cell.

Later, Tanaka et al. [45] extended ConSORT to support pointer arithmetic, enabling the verification of programs that manipulate integer arrays and perform pointer arithmetic. Their key idea is to extend the notion of ownership to *(fractional) ownership functions* from array indices to fractional ownerships. With this extension, their type system can reason about the properties of array elements in an index-sensitive way. Pointer arithmetic is reflected as index-shifting operations over the ownership function at the type system's level.

In this paper, we extend Tanaka et al.'s type system to handle another kind of programs that they do not support: programs manipulating *nested arrays*, which frequently arise in programs that manipulate matrices and tensors. Figure 1 is the motivating example that we use to illustrate our contribution. The function `initMatrix` takes an integer n and a pointer pp to an array of length n, and allocates an integer array to each element of the array pointed to by pp so that pp points to a matrix represented by a two-dimensional array; concretely, the matrix is initialized so that $pp + i$ points to an array with length $n - i$, all of whose elements are set to n. To this end, if $n > 0$, `initMatrix` allocates an array of length n, binds s to a pointer to it (Line 13), calls `initArray`(n, n, s) that initializes all the elements of the array pointed to by s of length n to n (Line 14), writes s to the memory cell pointed to by pp (Line 15), and recursively calls `initMatrix`$(n - 1, pp + 1)$ (Line 16).

```
1  initArray (l, n, p){
2    if l <= 0 then { 0 }
3    else {
4      p := n; // Updating the memory cell pointed to by p to n
5      let q = p + 1 in
6      let x = initArray(l-1, n, q) in 0
7    }
8  }
9
10 initMatrix(n, pp)
11 {
12   if n <= 0 then { 0 } else {
13     let s = alloc n : int ref in
14     let d = initArray(n, n, s) in
15     pp := s;
16     let d2 = initMatrix(n-1, pp+1) in 0
17   }
18 }
```

Figure 1 A motivating example.

Verifying the above specification of `initMatrix` using the idea of ownership functions by Tanaka et al. requires the ownership of an inner array to be dependent not only on its index but also on the index of the outer array. However, Tanaka et al.'s type system does not support such dependency; hence, their type system cannot handle the program in Figure 1.

Our extension, based on Tanaka et al.'s type system, introduces *(fractional) ownership terms* to describe a function from array indices to ownerships. Intuitively, the ownership of the inner array of the matrix pointed to by pp after the execution of `initMatrix` must satisfy the following: Via pointer pp, the memory cell $(*(pp + x_2)) + x_1$ can be updated if $0 \leq x_2 \leq n-1 \land 0 \leq x_1 \leq n-x_2-1$, where x_1 represents the index of an inner array and x_2 represents the index of an outer array. Using an ownership term, this constraint is described as $(0 \leq x_2 \leq n-1 \land 0 \leq x_1 \leq n-x_2-1) \Rightarrow 1$; here, $\Rightarrow 1$ expresses the ownership for reading and updating the memory cell. Note that this ownership term for an inner array refers to the index x_2 of the outer array.

Contribution. We formally define an imperative language and its type system equipped with ownership terms. We also prove the soundness of the type system: a well-typed program does not cause assertion failures, out-of-bounds array accesses, or null-pointer dereferences. We also design a type inference procedure and implement a prototype verifier based on the procedure. We apply our verifier to several programs that manipulate nested arrays, and we demonstrate that it successfully and efficiently verifies the functional correctness of these programs, which was not possible with Tanaka et al.'s type system. Although our motivating examples primarily use two-dimensional arrays for clarity of presentation, our type system and implementation support nested arrays of arbitrary depth. In particular, we demonstrate this generality by verifying programs that manipulate three- and four-dimensional arrays (Section 5).

The rest of this paper is organized as follows. Section 2 defines the target imperative language and its operational semantics. Section 3 presents the proposed type system and states its soundness. Section 4 describes a template-based type inference procedure. Section 5 reports on the experimental results, comparing the performance of our implemented verifier and evaluating the verification capabilities for programs involving nested arrays. Section 6 discusses the applicability of our approach to mainstream programming languages and possible

$$
\begin{array}{lcl}
\tau^- & ::= & \mathbf{int} \mid \tau^- \, \mathbf{ref} \\
e & ::= & x \mid \mathbf{let}\, x = n \,\mathbf{in}\, e \mid \mathbf{let}\, x = \mathbf{null} \,\mathbf{in}\, e \mid \mathbf{let}\, x = y - z \,\mathbf{in}\, e \\
& \mid & \mathbf{ifnp}\, x \,\mathbf{then}\, e_0 \,\mathbf{else}\, e_1 \mid \mathbf{let}\, x = f(y_1, \ldots, y_n) \,\mathbf{in}\, e \mid \mathbf{let}\, x = e_0 \,\mathbf{in}\, e_1 \\
& \mid & \mathbf{let}\, x = \mathbf{alloc}\, y : \tau^- \,\mathbf{ref}\,\mathbf{in}\, e \mid \mathbf{let}\, x = *y \,\mathbf{in}\, e \mid x := y \,;\, e \mid \mathbf{let}\, x = y \boxplus z \,\mathbf{in}\, e \\
& \mid & \mathbf{alias}(x = y \boxplus z) \,;\, e \mid \mathbf{alias}(x = *y) \,;\, e \mid \mathbf{assert}(\varphi) \,;\, e \\
fd & ::= & f \mapsto (x_1, \ldots, x_n)\, e \\
D & ::= & \{fd_1, \ldots, fd_n\} \\
P & ::= & \langle D, e \rangle
\end{array}
$$

Figure 2 Syntax of the target language.

extensions to more general heap structures. Section 7 describes related work. Section 8 concludes the paper. We omit some definitions and proofs, which will be found in a full version of the paper [15].

2 Target Language

2.1 Syntax

Figure 2 shows the syntax of the target language. The set of variables **Var** is ranged over by w, x, y, and z. The metavariable n represents integer constants; f represents function names; φ represents first-order logic formulae over integers. The metavariable e represents expressions; fd represents function definitions; D represents a finite set of function definitions; P represents programs; and τ^- represents simple types, consisting of integer and reference types. An expression of the form $\mathbf{let}\, x = \ldots \,\mathbf{in}\, e_1$ binds x in e_1. We write $[y_1/x_1, \ldots, y_n/x_n]$ for simultaneous capture-avoiding substitution of variable y_i for variable x_i (for $1 \leq i \leq n$). A substitution is also denoted by θ.

In this language, constants and the value of every intermediate computation are named with **let**. We informally explain the meaning of expression forms.

- $\mathbf{ifnp}\, x \,\mathbf{then}\, e_0 \,\mathbf{else}\, e_1$ evaluates e_0 if x is not a positive integer; it evaluates e_1 otherwise.
- $\mathbf{let}\, x = f(y_1, \ldots, y_n) \,\mathbf{in}\, e$ calls function f with actual arguments $y_1, \ldots, y_n$, binds x to the value returned from f, and evaluates e. Without loss of generality, we assume $y_i \neq y_j$ if $i \neq j$.
- $\mathbf{let}\, x = \mathbf{alloc}\, y : \tau^- \,\mathbf{ref}\,\mathbf{in}\, e$ allocates an array of size y, binds x to the pointer to the 0-th memory cell of the array, and evaluates e. We assume that the simple type of x (i.e., $\tau^- \,\mathbf{ref}$) is annotated. As shown below, the type annotation determines the initial values stored in the allocated array. If a nested **ref** type is specified as τ^-, then x is bound to a pointer to an array of length y whose elements are initialized to **null**. Each cell is expected to be initialized with a pointer to another (nested) array before it is accessed. If y is not positive, an empty array will be allocated and a dangling pointer will be returned. However, the type system prevents such a pointer from being accessed.
- $\mathbf{let}\, x = *y \,\mathbf{in}\, e$ reads from the memory cell pointed to by the pointer y, binds x to the read value, and evaluates e.
- $x := y \,;\, e$ updates the memory cell pointed to by x with the value y and evaluates e.
- $\mathbf{let}\, x = y \boxplus z \,\mathbf{in}\, e$ is for pointer arithmetic; it binds x to the pointer obtained by shifting y by offset z and evaluates e.

- **alias**$(x = y \boxplus z)\,;e$ (resp., **alias**$(x = *y)\,;e$) evaluates e if $x = y \boxplus z$ (resp., $x = *y$) holds; otherwise, it raises an exception indicating that the alias relationship does not hold. Operationally, this expression works as a runtime check for the alias relationship; if the alias relationship does not hold, execution is safely terminated by raising an exception. Statically, this expression serves as a hint to the type checker: it declares the programmer's intended aliasing relationship between pointers; the type checker is allowed to redistribute the ownership between the pointers x and $y \boxplus z$ (resp., $*y$); see T-AliasAddPtr and T-AliasDeref in Section 3.3 for details. As in previous work on fractional ownership [45, 47, 43, 42], our type system guarantees the absence of errors due to runtime assertion failures, out-of-bounds accesses, and null-pointer dereferences; an incorrect alias expression leads to the explicit error state **AliasFail** rather than to undefined behavior (see Section 2.2 and Theorem 3.1).
- **assert**$(\varphi)\,;e$ evaluates e if the formula φ holds; otherwise, the evaluation gets stuck due to an assertion failure, which means that the program goes into an unsafe state. A well-typed program in our type system is guaranteed not to cause this error.

A function definition fd is of the form $f \mapsto (x_1, \ldots, x_n)e$ where f is the name of the defined function, $x_1, \ldots, x_n$ are the names of the parameters, and e is the body of the function. We assume that $x_1, \ldots, x_n$ are pairwise distinct. Finally, a program P is a pair of a set of function definitions D and a main expression e.

2.2 Operational Semantics

We define the operational semantics of the language.

First, we give the syntax of *pointer values* and *values*, ranged over by pv and v, respectively.

$$a \in \mathcal{A} \qquad pv ::= (a, i) \mid \textbf{null} \qquad v ::= n \mid pv$$

An *address* is represented by a pair (a, i) of a base address a, which is an element of a fixed set $\mathcal{A}$, and an offset i, which is an integer. Two addresses (a, i) and (a', i') are equal if and only if $a = a'$ and $i = i'$. A *pointer value* pv is either an address or the null pointer **null**. Unlike other type systems using fractional ownership [43], in which **null** is used to represent the end of a linked list, our type system gives **null** zero ownership, preventing access to the null pointer statically. A *value* v is either a pointer value or an integer.

The small-step operational semantics of our language is given as a rewriting relation between configurations, ranged over by C, of the form $\langle R, H, e\rangle$ or an error state **AliasFail**. Here, R is a map from a finite subset of **Var** to the set of values, modeling a register file, and H is a (partial) map from the set of addresses to the set of values, modeling a heap. We write $dom(R)$ and $dom(H)$ for the domain of R and H, respectively. If $\{x_1, \ldots, x_n\} \cap dom(R) = \emptyset$, we write $R\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ for the map obtained by extending R with new bindings that map x_i to v_i (for $1 \le i \le n$). Similarly, if $\{(a, 0), \ldots, (a, n)\} \cap dom(H) = \emptyset$, we write $H\{(a, 0) \mapsto v_0, \ldots, (a, n) \mapsto v_n\}$ for an extension of H with $(a, i) \mapsto v_i$ (for $0 \le i \le n$) and, if $(a, n) \notin dom(H)$, we write $H\{(a, n) \hookleftarrow v\}$ for the heap that is identical to H except that the address (a, n) is now mapped to the value v.

Figure 3 gives the excerpt of the rules to define the reduction step $\longrightarrow_D$, parameterized by a set of function definitions D. The full set of rules is given in the extended version. We write $\mathbb{Z}$ for the set of integers.

The rules formalize the informal meaning described in Section 2.1 in a straightforward manner.

$$\frac{x' \notin dom(R)}{\langle R, H, \mathbf{let}\, x = \mathbf{null}\, \mathbf{in}\, e\rangle \longrightarrow_D \left\langle R\{x' \mapsto \mathbf{null}\}, H, [x'/x]e\right\rangle} \text{ (Rs-LetNull)}$$

$$\frac{R(y) = pv \qquad x' \notin dom(R) \qquad R(z) \in \mathbb{Z}}{\langle R, H, \mathbf{let}\, x = y \boxplus z\, \mathbf{in}\, e\rangle \longrightarrow_D \left\langle R\{x' \mapsto pv \boxplus R(z)\}, H, [x'/x]e\right\rangle} \text{ (Rs-AddPtr)}$$

$$\frac{(a,0) \notin dom(H) \qquad x' \notin dom(R) \qquad H' = H\{(a,0), \ldots, (a, R(y)\ -\ 1) \mapsto 0\}}{\langle R, H, \mathbf{let}\, x = \mathbf{alloc}\, y : \mathbf{int}\, \mathbf{ref}\, \mathbf{in}\, e\rangle \longrightarrow_D \left\langle R\{x' \mapsto (a,0)\}, H', [x'/x]e\right\rangle} \text{ (Rs-MkArrayIntref)}$$

$$\frac{(a,0) \notin dom(H) \qquad x' \notin dom(R) \qquad H' = H\{(a,0), \ldots, (a, R(y)\ -\ 1) \mapsto \mathbf{null}\} \qquad R(y) \in \mathbb{Z}}{\left\langle R, H, \mathbf{let}\, x = \mathbf{alloc}\, y : (\tau^-\ \mathbf{ref})\, \mathbf{ref}\, \mathbf{in}\, e\right\rangle \longrightarrow_D \left\langle R\{x' \mapsto (a,0)\}, H', [x'/x]e\right\rangle} \text{ (Rs-MkArrayNestedref)}$$

$$\frac{R(y) = pv \qquad R(z) \in \mathbb{Z} \qquad R(x) = pv \boxplus R(z)}{\langle R, H, \mathbf{alias}(x = y \boxplus z)\,; e\rangle \longrightarrow_D \langle R, H, e\rangle} \text{ (Rs-AliasAddPtr)}$$

$$\frac{H(R(y)) = R(x)}{\langle R, H, \mathbf{alias}(x = *y)\,; e\rangle \longrightarrow_D \langle R, H, e\rangle} \text{ (Rs-AliasDeref)}$$

$$\frac{R(y) = pv \qquad R(z) \in \mathbb{Z} \qquad R(x) \neq pv \boxplus R(z)}{\langle R, H, \mathbf{alias}(x = y \boxplus z)\,; e\rangle \longrightarrow_D \mathbf{AliasFail}} \text{ (Rs-AliasAddPtrFail)}$$

$$\frac{R(y) = \mathbf{null} \text{ or } H(R(y)) \neq R(x)}{\langle R, H, \mathbf{alias}(x = *y)\,; e\rangle \longrightarrow_D \mathbf{AliasFail}} \text{ (Rs-AliasDerefFail)}$$

$$\frac{\models [R]\,\varphi}{\langle R, H, \mathbf{assert}(\varphi)\,; e\rangle \longrightarrow_D \langle R, H, e\rangle} \text{ (Rs-Assert)}$$

Figure 3 Operational semantics (excerpt).

- The rules for evaluating an expression e that binds a variable x take a fresh variable x', extend the register file R with the value of the right-hand side, and rename the occurrences of x in the body of e to x' to avoid the collision of bound variable names.
- In the rule Rs-AddPtr for pointer arithmetic expressions $\mathbf{let}\, x = y \boxplus z\, \mathbf{in}\, e$, we use the meta-level operator $pv \boxplus j$, defined by: $(a, i) \boxplus j = (a, i + j)$ and $\mathbf{null} \boxplus j = \mathbf{null}$. Adding an offset to **null** results in **null**.
- The rules Rs-MkArrayIntref and Rs-MkArrayNestedref are for allocation $\mathbf{let}\, x = \mathbf{alloc}\, y : \tau^-\ \mathbf{ref}\, \mathbf{in}\, e$. In both rules a is a fresh base address and the heap H is extended to H' so that $(a,0), \ldots, (a, R(y) - 1)$ are all mapped to initial values that depend on τ^-. If τ^- is **int**, the initial values are 0 (Rs-MkArrayIntref); otherwise the initial values are **null** (Rs-MkArrayNestedref). In the latter case, each cell is intended to be initialized later with a pointer to another (nested) array before it is accessed. If $R(y) \leq 0$, we regard $\{(a,0) \mapsto 0, \ldots, (a, R(y) - 1) \mapsto 0\}$ as the empty set, hence $H' = H$: Although a fresh address is returned, no memory cell will be allocated. The operational semantics in Tanaka et al. [45] does not include a rule for nested array allocation.
- The two rules Rs-AliasAddPtr and Rs-AliasDeref deal with **alias** expressions. If the two pointers compared are equal, the execution proceeds to e without changing R or H. Otherwise, the configuration goes to **AliasFail**, which is a special erroneous configuration.
- The rule Rs-Assert requires φ to be valid under the assignment to variables according to R. $[R]\,\varphi$ is the first-order formula obtained by substituting $R(x)$ for each free variable x in φ if $R(x)$ is an integer. Formally, $[R]\,\varphi$ is defined as follows: $[R]\,\varphi = [R(x_1)/x_1, \ldots, R(x_n)/x_n]\,\varphi$ where $\{x_1, \ldots, x_n\} = dom(R)$. (In a well-typed program, all free variables in φ have the integer type.) If $[R]\,\varphi$ is not valid, the program gets stuck.

$$
\begin{array}{lllll}
\tau & \text{(types)} & ::= & \{\nu : \mathbf{int} \mid \varphi\} \mid \Pi x.(\tau\, \mathbf{ref}^{r}) \\
r & \text{(ownership terms)} & ::= & \varphi \Rightarrow q, r \mid \mathit{true} \Rightarrow q \\
\Gamma & \text{(type environments)} & ::= & x_1 : \tau_1, \ldots, x_n : \tau_n \\
\sigma & \text{(function types)} & ::= & \langle \Gamma \rangle \to \langle \Gamma' \mid \tau \rangle \quad (\text{where } \mathit{dom}(\Gamma) = \mathit{dom}(\Gamma'))
\end{array}
$$

Figure 4 Syntax of types.

A program $\langle D, e\rangle$ starts its execution from the configuration $\langle \emptyset, \emptyset, e\rangle$ and reduces by using $\longrightarrow_D$. The execution of an (untyped) program (1) terminates at a configuration of the form $\langle R, H, x\rangle$, representing successful termination, (2) diverges, (3) abnormally terminates at **AliasFail**, or (4) gets stuck. There are several reasons for an execution to get stuck.

1. Unbound variable errors, which occur when the referenced variable is not in the domain of R.
2. Simple type errors, such as applying subtraction to a pointer, conditional branching on a pointer, dereferencing an integer, and so on.
3. Dereferencing the null pointer. It occurs if $R(x)$ is **null** for a given $\langle R, H, \mathbf{let}\, y = *x\, \mathbf{in}\, e\rangle$ or $\langle R, H, x := y\,; e\rangle$.
4. An out-of-bounds array access. It occurs when $R(x) = (a, i) \notin \mathit{dom}(H)$, for a given $\langle R, H, \mathbf{let}\, y = *x\, \mathbf{in}\, e\rangle$ or $\langle R, H, x := y\,; e\rangle$.
5. An assertion failure. This happens when $\langle R, H, \mathbf{assert}(\varphi)\,; e\rangle$ does not satisfy $\models [R]\,\varphi$.

Our type system guarantees that well-typed programs do not get stuck.

3 Type System

In this section, we give our type system to prevent assertion failures and null/dangling pointer access and state a type soundness theorem. The type system is based on Tanaka et al. [45]; we give a more precise formalization of ownership functions, in particular, how they depend on the variables in context.

3.1 Types

The syntax of *types*, ranged over by τ, *ownership terms*, ranged over by r, and *function types*, ranged over by σ is shown in Figure 4. An integer type $\{\nu : \mathbf{int} \mid \varphi\}$ represents integers satisfying a refinement predicate φ. Precisely, $\{\nu : \mathbf{int} \mid \varphi\}$ stands for a subset of integers i that satisfy $[i/\nu]\varphi$. We often write **int** for $\{\nu : \mathbf{int} \mid \top\}$.

A type of the form $\Pi x.(\tau\, \mathbf{ref}^{r})$ represents pointers to arrays of type τ with an ownership term r, explained below. The variable x, which stands for an array index, is bound in τ and r. Thus, the element type and ownership term can depend on the array index. Moreover, if a reference type is nested, the inner type can depend on outer array indices. We write $|\tau|$ for the simple type obtained from τ in an obvious manner. For example, let $\tau_{ex} := \Pi x_1.(\Pi x_2.(\Pi x_3.(\{\nu : \mathbf{int} \mid \varphi\}\, \mathbf{ref}^{r_3})\, \mathbf{ref}^{r_2})\, \mathbf{ref}^{r_1})$ be a type for three-dimensional arrays of integers with ownership terms r_1, r_2, and r_3, where x_1 is the index of the outermost array, x_2 that of the middle array, and x_3 that of the innermost array. The index x_1 may appear in r_1, r_2, r_3, and φ; x_2 may appear in r_2, r_3, and φ; and x_3 may appear in r_3 and φ.

As in previous work [45, 47], the type system keeps track of pointer *ownership*, which is represented by a rational number q between 0 and 1 (that is, $(0 \le q \le 1)$). The rational number denotes the permissions for dereferencing (reading) and writing to the element. The ownership expresses permissions granted to pointers as follows:

- $q = 1$: allows both dereferencing and writing to the element (read-write permission).
- $0 < q < 1$: allows dereferencing but not writing (read-only permission).
- $q = 0$: allows neither reading nor writing (no access).[1]

An ownership term, which takes the form $\varphi_1 \Rightarrow q_1, \ldots, \varphi_n \Rightarrow q_n, \mathit{true} \Rightarrow q_0$ where each q_i is a rational number such that $0 \leq q_i \leq 1$, represents the ownership of the pointer to each element of an array. Intuitively, it means "if φ_1 is true, then the ownership is q_1; or else if φ_2 is true, then the ownership is $q_2, \ldots$". For example, the type $\Pi x.(\{\nu : \mathbf{int} \mid \nu > x\}\, \mathbf{ref}^{(\varphi \Rightarrow 1, \mathit{true} \Rightarrow 0)})$ where $\varphi = 0 \leq x \leq 2$ means that the pointer has full ownership of the first three elements but no ownership of the other elements, and that the i-th element is an integer greater than i. An ownership term always ends with $\mathit{true} \Rightarrow q$, ensuring at least one case applies. We write $\mathbf{0}$ and $\mathbf{1}$ for $\mathit{true} \Rightarrow 0$ and $\mathit{true} \Rightarrow 1$, respectively, for brevity. We even write $\varphi \Rightarrow q$ for $\varphi \Rightarrow q, \mathbf{0}$ – when the default case returns 0. For example, in the above type τ_{ex} for three-dimensional arrays, if $r_1 = (x_1 \in [0, 9] \Rightarrow 1, \mathbf{0})$, $r_2 = (x_2 \in [0, x_1] \Rightarrow 1, \mathbf{0})$, and $r_3 = (x_3 \in [0, x_2] \Rightarrow 1, \mathit{true} \Rightarrow 0)$, then the pointer has full ownership at indices i_1, i_2, i_3, where i_1, i_2, i_3 range over the outermost, middle, and innermost indices, and satisfy $0 \leq i_3 \leq i_2 \leq i_1 \leq 9$.

Type Environments. A *type environment*, denoted by Γ, is a sequence of type declarations $x_i : \tau_i$ of pairwise distinct variables. Given a type environment $x_1 : \tau_1, .., x_n : \tau_n$, all the variables are bound in $\tau_1, \ldots, \tau_n$. Thus, even mutual dependency between variables is allowed. We regard a type environment $x_1 : \tau_1, .., x_n : \tau_n$ as a function that maps each x_i to τ_i. We write $\mathit{dom}(\Gamma)$ for $\{x_1, \ldots, x_n\}$. If $x \notin \mathit{dom}(\Gamma)$, we write $\Gamma, x : \tau$ to add a new declaration $x : \tau$ to Γ. We write $\Gamma[x : \tau]$ to signify $\Gamma(x) = \tau$ and, if $x \in \mathit{dom}(\Gamma)$, write $\Gamma\,[x \hookleftarrow \tau]$ for a type environment Γ' such that $\mathit{dom}(\Gamma') = \mathit{dom}(\Gamma)$ and $\Gamma'(x) = \tau$ and $\Gamma'(y) = \Gamma(y)$ for $y \neq x$.

Function Types and Function Type Environments. A *function type* σ is of the form $\langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle \to \langle x_1 : \tau'_1, \ldots, x_n : \tau'_n \mid \tau \rangle$. It means that the function takes $x_1, \ldots, x_n$ of types $\tau_1, \ldots, \tau_n$, respectively, as arguments, returns τ, and changes the types of the arguments to $\tau'_1, \ldots, \tau'_n$ as a side effect. Formal arguments are named and can appear in τ_i, τ'_i, and τ. For example, the function type

$$\langle x_1 : \mathbf{int}, x_2 : \Pi z.(\mathbf{int}\, \mathbf{ref}^{r}) \rangle \to \langle x_1 : \mathbf{int}, x_2 : \Pi z.(\{\nu : \mathbf{int} \mid \nu = x_1\}\, \mathbf{ref}^{r}) \mid \mathbf{int} \rangle$$

where $r = (0 \leq z \leq x_1 \Rightarrow 1)$ means that the function takes an integer x_1 and an integer array whose length is x_1, returns an integer, and updates the elements of the array with x_1.

A *function type environment*, denoted by Θ, is a finite mapping from function names to function types.

Type Well-Formedness. We enforce well-formedness conditions on types. Roughly speaking, a type τ is well-formed under type environment Γ, written $\Gamma \vdash \tau$ ok, if every predicate φ in τ refers only to integer variables in context and if the ownership of an array element is zero, its element type is "empty". Intuitively, a type is empty if all ownership in it is *semantically* $\mathbf{0}$. For example, the type $\tau = \Pi x.(\Pi y.(\mathbf{int}\, \mathbf{ref}^{r_0})\, \mathbf{ref}^{r})$ where $r = (0 \leq x \leq 1 \Rightarrow 1)$ and $r_0 = (0 \leq x \leq 1 \wedge 0 \leq y \leq 1 \Rightarrow 1)$ is well-formed but $\tau' = \Pi x.(\Pi y.(\mathbf{int}\, \mathbf{ref}^{r'_0})\, \mathbf{ref}^{r})$ with

[1] Toman et al. [47] *do* allow reading even if $q = 0$. Their type system does not support pointer arithmetic or prevent null pointer access. Thus, every pointer access is safe (except for the null pointer). The 0 ownership is different from positive ownership in that the refinement predicate of the element pointed to by a pointer with ownership 0 is regarded as $\top$, because the ownership being 0 means that there may be a writable alias with ownership 1.

$r'_0 = (0 \le x \le 2 \land 0 \le y \le 1 \Rightarrow 1)$ is not. In τ, the ownership of the element of an inner array is 1 only if the outer array element is accessible (i.e., x is either 0 or 1), whereas, in τ', the ownership of an element of the third inner array is 1 even if the outer array is not accessible, violating the emptiness condition.

A type environment Γ is well-formed if each type declared in Γ is well-formed under Γ itself. A function type $\langle\Gamma\rangle \to \langle\Gamma' \mid \tau\rangle$ is well-formed if both Γ and Γ' are well-formed, and τ is well-formed under Γ'; a function type environment is well-formed if each function type in it is well-formed.

We will assume that types, type environments, function types, and function type environments are well-formed in judgments introduced later and omit well-formedness conditions from derivation rules. The type environment under which a given type is well-formed is obvious in most cases – we will note when it is not clear.

Remark about Tanaka et al. In Tanaka et al. [45], an array type is decorated with an *ownership function* r, which is a (seemingly set-theoretic) *function* from integers to rational numbers (in the interval $[0, 1]$). The way ownership information is represented there, however, obscures how reference types may depend on array indices and other variables, especially when they are nested.

For example, the type of `pp` on Line 16 in Figure 1 is $\Pi x_1.(\Pi x_2.(\{\nu : \mathbf{int} \mid \varphi\}\,\mathbf{ref}^{r_2})\,\mathbf{ref}^{r_1})$ where the ownership term r_2 of the inner array type is $0 \le x_2 \le \mathtt{n} - x_1 \Rightarrow 1$ since the length of the array pointed to by $\mathtt{pp} \boxplus i$ is $\mathtt{n} - i$. Because r_2 depends on x_1, it would need to be a *family* of functions, rather than a function.

3.2 Auxiliary Judgments and Encoding into Verification Logic

We often encode auxiliary judgments into formulae within the logic used for verification. For example, we write $\Gamma \models \varphi$, which means that formula φ is valid under Γ. This relation is formally defined as follows: $\Gamma \models \varphi \iff \models \mathit{fml}(\Gamma) \implies \varphi$, where the function $\mathit{fml}(\Gamma)$ is defined by:

$$\mathit{fml}(x_1 : \tau_1, \ldots, x_n : \tau_n) = \mathit{fml}_{x_1}(\tau_1) \land \cdots \land \mathit{fml}_{x_n}(\tau_n)$$
$$\mathit{fml}_y(\{\nu : \mathbf{int} \mid \varphi\}) = [y/\nu]\varphi \qquad \mathit{fml}_y(\Pi x.(\tau'\,\mathbf{ref}^r)) = \top.$$

The function $\mathit{fml}(\Gamma)$ translates refinements on integer variables into propositions, ignoring variables of reference types.

We also encode auxiliary judgments for pointwise comparison and addition of ownership functions, such as $\Gamma \models r_1 \le r_2$ and $\Gamma \models r_1 + r_2 = r_3$ into the verification logic. Intuitively, the comparisons $r_1 \le r_2$ and $r_1 = r_2$ for ownership terms are defined as a pointwise extension of comparison over rational numbers. We denote the first-order formula expressing $r_1 \le r_2$ by $\mathit{fml}(r_1 \le r_2)$. We omit the full definition for brevity. For example, if $r_1 = (\varphi_1 \Rightarrow q_{11}, \mathit{true} \Rightarrow q_{12})$ and $r_2 = (\varphi_2 \Rightarrow q_{21}, \mathit{true} \Rightarrow q_{22})$, then $\mathit{fml}(r_1 \le r_2)$ is

$$(\varphi_1 \land \varphi_2 \implies q_{11} \le q_{21}) \land (\neg\varphi_1 \land \varphi_2 \implies q_{12} \le q_{21})$$
$$\land (\varphi_1 \land \neg\varphi_2 \implies q_{11} \le q_{22}) \land (\neg\varphi_1 \land \neg\varphi_2 \implies q_{12} \le q_{22})$$

where $q_{ij} \le q_{kl}$ stands for either $\top$ or $\bot$, depending on the comparison of the two rational numbers. (Note that no rational number will appear in the resulting formula because the comparison is performed during the encoding.) Then, $\Gamma \models r_1 \le r_2$ stands for $\models \mathit{fml}(\Gamma) \implies \mathit{fml}(r_1 \le r_2)$. $\mathit{fml}(r_1 = r_2)$ and the encoding of pointwise addition $\mathit{fml}(r_1 + r_2 = r_3)$ can be defined similarly.

$\boxed{\Theta \mid \Gamma \vdash e : \tau \Rightarrow \Gamma'}$

$$\frac{\Gamma \vdash \tau_1 + \tau_2 \approx \tau_3}{\Theta \mid \Gamma[x : \tau_3] \vdash x : \tau_1 \Rightarrow \Gamma\,[x \hookleftarrow \tau_2]} \quad \text{(T-Var)}$$

$$\frac{\Gamma \models \mathit{Empty}(\Pi z.(\tau\,\mathbf{ref}^r)) \qquad \Theta \mid \Gamma, x : \Pi z.(\tau\,\mathbf{ref}^r) \vdash e : \tau \Rightarrow (\Gamma', x : \tau')}{\Theta \mid \Gamma \vdash \mathbf{let}\, x = \mathbf{null}\, \mathbf{in}\, e : \tau \Rightarrow \Gamma'} \quad \text{(T-Null)}$$

$$\frac{\begin{array}{c}\Theta \mid \Gamma\,[x \hookleftarrow \{\nu : \mathbf{int} \mid \varphi \wedge \nu \leq 0\}] \vdash e_0 : \tau \Rightarrow \Gamma' \\ \Theta \mid \Gamma\,[x \hookleftarrow \{\nu : \mathbf{int} \mid \varphi \wedge \nu > 0\}] \vdash e_1 : \tau \Rightarrow \Gamma'\end{array}}{\Theta \mid \Gamma[x : \{\nu : \mathbf{int} \mid \varphi\}] \vdash \mathbf{ifnp}\, x\, \mathbf{then}\, e_0\, \mathbf{else}\, e_1 : \tau \Rightarrow \Gamma'} \quad \text{(T-If)}$$

$$\frac{\begin{array}{c}\Theta(f) = \langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle \to \langle x_1 : \tau_1', \ldots, x_n : \tau_n' \mid \tau \rangle \qquad \theta = [y_1/x_1, \ldots, y_n/x_n] \\ \Theta \mid \Gamma\,[y_i \hookleftarrow \theta\,\tau_i'], x : \theta\,\tau \vdash e : \tau' \Rightarrow (\Gamma', x : \tau'')\end{array}}{\Theta \mid \Gamma[y_i : \theta\,\tau_i] \vdash \mathbf{let}\, x = f(y_1, \ldots, y_n)\, \mathbf{in}\, e : \tau' \Rightarrow \Gamma'} \quad \text{(T-Call)}$$

$$\frac{\Gamma \leq \Gamma' \qquad \Theta \mid \Gamma' \vdash e : \tau \Rightarrow \Gamma'' \qquad \Gamma'', \tau \leq \Gamma''', \tau'}{\Theta \mid \Gamma \vdash e : \tau' \Rightarrow \Gamma'''} \quad \text{(T-Sub)}$$

Figure 5 Expression typing rules (1).

3.3 Type System

Our type system consists of several judgment forms. The main one is for expression typing of the form: $\Theta \mid \Gamma \vdash e : \tau \Rightarrow \Gamma'$. Intuitively, it means "under the environments Θ and Γ, the type of expression e is τ, and the initial type environment Γ is updated to Γ'." As we already noted, we assume the type environments and the type are well-formed, namely $\vdash \Gamma$ ok and $\vdash \Gamma'$ ok and $\Gamma' \vdash \tau$ ok. We often call Γ a pre-environment and Γ' a post-environment.

Figures 5 and 6 show the typing rules for expressions. The rules in Figure 5 are for expressions not related to heap manipulation and are very similar to those in previous work [45, 47], except for a few minor notational changes. Standard typing rules (i.e., T-Int, T-Minus, and T-Let) are omitted from Figure 5 for brevity; see the full version.

In T-Null, the type of x is given an *empty* reference type because **null** is inaccessible. The premise $\Gamma \models \mathit{Empty}(\tau)$ intuitively means that τ is empty under Γ. Similarly to $r_1 \leq r_2$, the type emptiness is also encoded into a logical formula. The function $\mathit{Empty}(\tau)$ is defined by induction on τ as follows:

$$\begin{array}{rcl}\mathit{Empty}(\{\nu : \mathbf{int} \mid \varphi\}) & = & \top \\ \mathit{Empty}(\Pi z.(\tau\,\mathbf{ref}^r)) & = & \mathit{fml}(r = \mathbf{0}) \wedge \mathit{Empty}(\tau).\end{array}$$

$\mathit{Empty}(\tau)$ gathers conditions recursively if τ is a nested array type.

The rule T-If is standard in a refinement type system. Since the **then** branch (the **else** branch, resp.) is taken only when x is non-positive (positive, resp.), the predicate φ for x is strengthened by $\nu \leq 0$ ($\nu > 0$, resp.).

In rule T-Call, substitution θ renames formal parameter names in the function type $\Theta(f)$ by actual argument names. This rule means that the types of arguments y_i change from $\theta\,\tau_i$ to $\theta\,\tau_i'$ by the function call, x is given type $\theta\,\tau$, where τ is the return type.

The rule T-Sub is for subsumption, which strengthens the pre-environment ($\Gamma \leq \Gamma'$) and weakens the expression type and the post-environment. The full rules for subtyping are similar to those of previous work [45, 47] and are given in the extended version. We explain two rules here. The rule S-Int for integer types:

$$\frac{\Gamma, \nu\colon \mathbf{int} \models \varphi_1 \implies \varphi_2}{\Gamma \vdash \{\nu : \mathbf{int} \mid \varphi_1\} \leq \{\nu : \mathbf{int} \mid \varphi_2\}} \quad \text{(S-Int)}$$

requires that the predicate of a subtype has to be stronger than that of a supertype. The rule S-Ref for reference types:

$$\frac{\Gamma, x\colon \mathbf{int} \models r_1 \geq r_2 \qquad \Gamma, x\colon \mathbf{int} \vdash \tau_1 \leq \tau_2}{\Gamma \vdash \Pi x.(\tau_1\, \mathbf{ref}^{r_1}) \leq \Pi x.(\tau_2\, \mathbf{ref}^{r_2})} \quad \text{(S-Ref)}$$

is covariant and additionally requires that the ownership term in a subtype has to be pointwise greater than that in a supertype. Note that covariance is safe because if an array is writable through one alias, the other aliases will be assigned no ownership and cannot observe the update [9, Section 6.3].

The rule T-Var is one of the key rules. To understand the rule, consider, for example, $\mathbf{let}\, y = x\, \mathbf{in}\, e_0$, which creates an alias of x. A part of the ownership of x has to be given to y when typing e_0. Such "split" of ownership is expressed as the auxiliary judgment $\Gamma \vdash \tau_1 + \tau_2 \approx \tau_3$ – which reads "τ_3 is split into τ_1 and τ_2 under Γ" or "τ_1 and τ_2 merge into τ_3 under Γ" – for type addition. The latter reading will be useful when we discuss the typing rules for **alias** expressions.

We show a type derivation for $\mathbf{let}\, y = x\, \mathbf{in}\, e_0$ below. Here, we write $\tau(r) = \Pi z.(\mathbf{int}\, \mathbf{ref}^r)$ and omit Θ.

$$\frac{\dfrac{\vdots}{\Gamma, x : \tau(r_1) \vdash x : \tau(r_2) \Rightarrow \Gamma, x : \tau(r_2)} \qquad \dfrac{\vdots}{\Gamma, x : \tau(r_2), y : \tau(r_2) \vdash e_0 : \tau(r_2) \Rightarrow (\Gamma', y : \tau(r_2))}}{\Gamma, x : \tau(r_1) \vdash \mathbf{let}\, y = x\, \mathbf{in}\, e : \tau_0 \Rightarrow \Gamma'}$$

where $r_1 = 0 \leq z \leq 9 \Rightarrow 1$ and $r_2 = 0 \leq z \leq 9 \Rightarrow 0.5$. Notice that $\Gamma \models r_2 + r_2 = r_1$ holds.

Now, we explain the typing rules related to heap-manipulating constructs.

The rules T-MkIntArray and T-MkNestedArray are for the creation of new arrays of integers and (nested) arrays, respectively. In both rules, the first premise about r (in the type of x) means that, x has full ownership $\Gamma \models r = \mathbf{1}$, i.e., write permission, for the 0-th y elements but no ownership $\Gamma \models r = \mathbf{0}$ for all the other elements. For integer arrays, the element type asserts that all (accessible) elements are initialized to 0. Since nested arrays are initialized with **null**, the element type τ' has to be *Empty*.

The rule T-Deref is for dereference $\mathbf{let}\, x = *y\, \mathbf{in}\, e$. The first premise requires the ownership at $z = 0$ is greater than 0, meaning that the address denoted by y (with offset 0) can be read. Since x becomes an alias of $*y$, the type of the 0th element must be split – analogously to variable reference – and distributed between x and y after the dereference. However, since the types of all elements have to be represented by a single type expression τ_y, which depends on the array index z, type splitting is more complicated. The second premise means that the type τ_y (under $z = 0$) for the 0-th element is split into τ' and τ_x, the latter of which is given to x. The third and fourth premises concern the element types τ'_y after dereferencing. They roughly mean that the 0-th element type (τ'_y with $z = 0$) is equivalent to τ' and the other element types (τ'_y with $z \neq 0$) remain the same as τ_y. We use another judgment $\Gamma \vdash \tau_1 \approx \tau_2$ to state the two types are *equivalent*, defined by: $\Gamma \vdash \tau_1 \approx \tau_2 \iff \Gamma \vdash \tau_1 \leq \tau_2$ and $\Gamma \vdash \tau_2 \leq \tau_1$. The type $(\tau')^{=x}$ in the third premise stands for the type obtained from τ' by adding the fact that $*y$ is equal to x, if the element type is integer. Formally, $(\tau)^{=x}$ is defined as follows:

$$\{\nu : \mathbf{int} \mid \varphi\}^{=x} = \{\nu : \mathbf{int} \mid \varphi \wedge (\nu = x)\} \qquad \Pi z.(\tau\, \mathbf{ref}^r)^{=x} = \Pi z.(\tau\, \mathbf{ref}^r)$$

$\boxed{\Theta \mid \Gamma \vdash e : \tau \Rightarrow \Gamma'}$

$$\frac{\begin{array}{c}\Gamma, z : \mathbf{int} \models r = ((0 \le z \wedge z \le y - 1) \Rightarrow 1) \\ \Theta \mid \Gamma[y : \{\nu : \mathbf{int} \mid \varphi\}], x : \Pi z.(\{\nu : \mathbf{int} \mid 0 \le z \wedge z \le y - 1 \implies \nu = 0\}\, \mathbf{ref}^{r}) \vdash e_0 : \tau \Rightarrow (\Gamma', x : \tau')\end{array}}{\Theta \mid \Gamma[y : \{\nu : \mathbf{int} \mid \varphi\}] \vdash \mathbf{let}\, x = \mathbf{alloc}\, y : \mathbf{int}\, \mathbf{ref}\, \mathbf{in}\, e_0 : \tau \Rightarrow \Gamma'} \text{ (T-MkIntArray)}$$

$$\frac{\begin{array}{c}\Gamma, z : \mathbf{int} \models r = ((0 \le z \wedge z \le y - 1) \Rightarrow 1) \qquad \Gamma, z : \mathbf{int} \models \mathit{Empty}(\tau') \\ \Theta \mid \Gamma[y : \{\nu : \mathbf{int} \mid \varphi\}], x : \Pi z.(\tau'\, \mathbf{ref}^{r}) \vdash e_0 : \tau \Rightarrow (\Gamma', x : \tau') \qquad |\tau'| = \tau^{-}\, \mathbf{ref}\end{array}}{\Theta \mid \Gamma[y : \{\nu : \mathbf{int} \mid \varphi\}] \vdash \mathbf{let}\, x = \mathbf{alloc}\, y : (\tau^{-}\, \mathbf{ref})\, \mathbf{ref}\, \mathbf{in}\, e_0 : \tau \Rightarrow \Gamma'} \text{ (T-MkNestedArray)}$$

$$\frac{\begin{array}{c}\Gamma, z : \{\nu : \mathbf{int} \mid \nu = 0\} \models r > \mathbf{0} \qquad \Gamma, z : \{\nu : \mathbf{int} \mid \nu = 0\} \vdash \tau' + \tau_x \approx \tau_y \\ \Gamma, x : \tau_x, z : \{\nu : \mathbf{int} \mid \nu = 0\} \vdash \tau'_y \approx (\tau')^{=x} \qquad \Gamma, x : \tau_x, z : \{\nu : \mathbf{int} \mid \nu \neq 0\} \vdash \tau'_y \approx \tau_y \\ \Theta \mid \Gamma\left[y \hookleftarrow \Pi z.(\tau'_y\, \mathbf{ref}^{r})\right], x : \tau_x \vdash e_0 : \tau \Rightarrow (\Gamma', x : \tau'')\end{array}}{\Theta \mid \Gamma[y : \Pi z.(\tau_y\, \mathbf{ref}^{r})] \vdash \mathbf{let}\, x = *y\, \mathbf{in}\, e_0 : \tau \Rightarrow \Gamma'} \text{ (T-Deref)}$$

$$\frac{\begin{array}{c}\Gamma, z : \{\nu : \mathbf{int} \mid \nu = 0\} \models r = \mathbf{1} \qquad \Gamma \vdash \tau' + \tau'_y \approx \tau_y \\ \Gamma, z : \{\nu : \mathbf{int} \mid \nu = 0\} \vdash \tau'_{*x} \approx (\tau')^{=y} \qquad \Gamma, z : \{\nu : \mathbf{int} \mid \nu \neq 0\} \vdash \tau'_{*x} \approx \tau_{*x} \\ \Theta \mid \Gamma\left[x \hookleftarrow \Pi z.(\tau'_{*x}\, \mathbf{ref}^{r})\right]\left[y \hookleftarrow \tau'_y\right] \vdash e_0 : \tau \Rightarrow \Gamma'\end{array}}{\Theta \mid \Gamma[x : \Pi z.(\tau_{*x}\, \mathbf{ref}^{r})][y : \tau_y] \vdash x := y\,; e_0 : \tau \Rightarrow \Gamma'} \text{ (T-Assign)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \Pi w.(\tau_1\, \mathbf{ref}^{r_{y_1}}) + \Pi w.[(w - z)/w](\tau_2\, \mathbf{ref}^{r_x}) \approx \Pi w.(\tau_3\, \mathbf{ref}^{r_y}) \\ \Theta \mid \Gamma[z : \{\nu : \mathbf{int} \mid \varphi\}]\left[y \hookleftarrow \Pi w.(\tau_1\, \mathbf{ref}^{r_{y_1}})\right], x : \Pi w.(\tau_2\, \mathbf{ref}^{r_x}) \vdash e_0 : \tau \Rightarrow (\Gamma', x : \tau')\end{array}}{\Theta \mid \Gamma[y : \Pi w.(\tau_3\, \mathbf{ref}^{r_y})][z : \{\nu : \mathbf{int} \mid \varphi\}] \vdash \mathbf{let}\, x = y \boxplus z\, \mathbf{in}\, e_0 : \tau \Rightarrow \Gamma'} \text{ (T-AddPtr)}$$

$$\frac{\begin{array}{c}\Gamma \vdash (\Pi w'.[(w' - z)/w'](\tau_{*x}\, \mathbf{ref}^{r_x}) + \Pi w.(\tau_{*y}\, \mathbf{ref}^{r_y})) \approx \\ (\Pi w'.[(w' - z)/w'](\tau'_{*x}\, \mathbf{ref}^{r'_x})) + \Pi w.(\tau'_{*y}\, \mathbf{ref}^{r'_y}) \\ \Theta \mid \Gamma[z : \{\nu : \mathbf{int} \mid \varphi\}]\left[x \hookleftarrow \Pi w'.(\tau'_{*x}\, \mathbf{ref}^{r'_x})\right]\left[y \hookleftarrow \Pi w.(\tau'_{*y}\, \mathbf{ref}^{r'_y})\right] \vdash e_0 : \tau \Rightarrow \Gamma'\end{array}}{\Theta \mid \Gamma[x : \Pi w'.(\tau_{*x}\, \mathbf{ref}^{r_x})][y : \Pi w.(\tau_{*y}\, \mathbf{ref}^{r_y})][z : \{\nu : \mathbf{int} \mid \varphi\}] \vdash \mathbf{alias}(x = y \boxplus z)\,; e_0 : \tau \Rightarrow \Gamma'} \text{ (T-AliasAddPtr)}$$

$$\frac{\begin{array}{c}\Gamma, w : \{\nu : \mathbf{int} \mid \nu = 0\} \vdash \\ (\Pi z'.(\tau_{*x}\, \mathbf{ref}^{r_x}) + \Pi z.(\tau_{**y}\, \mathbf{ref}^{r_{*y}})) \approx (\Pi z'.(\tau'_{*x}\, \mathbf{ref}^{r'_x}) + \Pi z.(\tau'_{**y}\, \mathbf{ref}^{r'_{*y}})) \\ \Gamma, w : \{\nu : \mathbf{int} \mid \nu \neq 0\} \vdash \Pi z.(\tau_{**y}\, \mathbf{ref}^{r_{*y}}) \approx \Pi z.(\tau'_{**y}\, \mathbf{ref}^{r'_{*y}}) \\ \Theta \mid \Gamma\left[x \hookleftarrow \Pi z'.(\tau'_{*x}\, \mathbf{ref}^{r'_x})\right]\left[y \hookleftarrow \Pi w.(\Pi z.(\tau'_{**y}\, \mathbf{ref}^{r'_{*y}})\, \mathbf{ref}^{r})\right] \vdash e_0 : \tau \Rightarrow \Gamma'\end{array}}{\Theta \mid \Gamma[x : \Pi z'.(\tau_{*x}\, \mathbf{ref}^{r_x})][y : \Pi w.(\Pi z.(\tau_{**y}\, \mathbf{ref}^{r_{*y}})\, \mathbf{ref}^{r})] \vdash \mathbf{alias}(x = *y)\,; e_0 : \tau \Rightarrow \Gamma'} \text{ (T-AliasDeref)}$$

$$\frac{\Gamma \models \varphi \qquad \Theta \mid \Gamma \vdash e_0 : \tau \Rightarrow \Gamma'}{\Theta \mid \Gamma \vdash \mathbf{assert}(\varphi)\,; e_0 : \tau \Rightarrow \Gamma'} \text{ (T-Assert)}$$

Figure 6 Expression typing rules (2).

```
1  // pp:Πx2(.Πx1.({ν:int | ⊤} ref^0)) ref^{0≤x2≤n−1 ⇒ 1}
2  let s = alloc n : int ref in
3  // s:(Πx1.{ν:int | 0 ≤ x1 ≤ n − 1 ⟹ ν = 0}) ref^{0≤x1≤n−1 ⇒ 1}
4  let d = initArray(s, n, n) in
5  // s:(Πx1.{ν:int | 0 ≤ x1 ≤ n − 1 ⟹ ν = n}) ref^{0≤x1≤n−1 ⇒ 1}
6  pp := s;
7  // pp:(Πx2.(Πx1.{ν:int | x2 = 0 ∧ 0 ≤ x1 ≤ n − 1 ⟹ ν = n}) ref^{r1}) ref^{r2}
8  // r1 = (x2 = 0 ∧ 0 ≤ x1 ≤ n − 1 ⇒ 1), r2 = (0 ≤ x2 ≤ n − 1 ⇒ 1)
9  // s:(Πx1.{ν:int | ⊤}) ref^0
10 let t = pp ⊞ 1 in
11 // pp:(Πx2.(Πx1.{ν:int | x2 = 0 ∧ 0 ≤ x1 ≤ n − 1 ⟹ ν = n}) ref^{r1}) ref^{r2}
12 // r1 = (x2 = 0 ∧ 0 ≤ x1 ≤ n − 1 ⇒ 1), r2 = (x2 = 0 ⇒ 1)
13 // t:(Πx2.(Πx1.{ν:int | ⊤}) ref^0) ref^{0≤x2≤n−2 ⇒ 1}
14 let d2 = initMatrix(n-1, t) in
15 // t:(Πx2.(Πx1.{ν:int | 0 ≤ x2 ≤ n − 2 ∧ 0 ≤ x1 ≤ n − x2 − 2 ⟹ ν = n}) ref^{r1}) ref^{r2}
16 // r1 = (0 ≤ x2 ≤ n − 2 ∧ 0 ≤ x1 ≤ n − x2 − 2 ⇒ 1), r2 = (0 ≤ x2 ≤ n − 2 ⇒ 1)
17 alias(t = pp + 1); 0
18 // pp:(Πx2.(Πx1.{ν:int | 0 ≤ x2 ≤ n − 1 ∧ 0 ≤ x1 ≤ n − x2 − 1 ⟹ ν = n}) ref^{r1}) ref^{r2}
19 // r1 = (0 ≤ x2 ≤ n − 1 ∧ 0 ≤ x1 ≤ n − x2 − 1 ⇒ 1), r2 = (0 ≤ x2 ≤ n − 1 ⇒ 1)
20 // t:(Πx2.(Πx1.{ν:int | ⊤}) ref^0) ref^0
```

Figure 7 Typing example.

The rule T-Assign handles updating an array element. The first premise requires the ownership at $z = 0$ is 1, meaning that the address denoted by y (with offset 0) is writable. The second, third, and fourth premises are similar to those in T-Deref. Here, the type of y is split into τ' and τ'_y. The former is used as the 0-th element type and the latter the type of y after the update.

The rule T-AddPtr for pointer arithmetic can be considered as a generalization of typing for $\mathbf{let}\, x = y \,\mathbf{in}\, e_0$ discussed above (except that the type of y is a reference type), in the sense that the type of y before **let** is split into the type for x and y after **let**. The capture-avoiding substitution $[w - z/w]$ (for logical terms for variables in formulae) means that the n-th element of y corresponds to the $(n - z)$-th element of x.

The rules T-AliasAddPtr and T-AliasDeref handle **alias** expressions. From the viewpoint of ownership, an **alias** expression merges and re-splits the types for the aliases into two new types by using type addition. The judgment $\Gamma \models \tau_1 + \tau_2 \approx \tau_3 + \tau_4$ stands for $\Gamma \models \tau_1 + \tau_2 \approx \tau_5$ and $\Gamma \models \tau_3 + \tau_4 \approx \tau_5$ for some τ_5. A typical use is to transfer the predicate of one alias to another: for example, we can derive

$$\Theta \mid \Gamma_1 \vdash \mathbf{alias}(x = y \boxplus z)\,; z : \mathbf{int} \Rightarrow \Gamma_2$$

where

$$\Gamma_1 = x : \Pi w.(\{\nu : \mathbf{int} \mid \nu = w\}\,\mathbf{ref}^{r_1}), y : \Pi w.(\mathbf{int}\,\mathbf{ref}^{\mathbf{0}}), z : \{\nu : \mathbf{int} \mid z = 1\},$$
$$\Gamma_2 = x : \Pi w.(\{\nu : \mathbf{int} \mid \nu = w\}\,\mathbf{ref}^{r_2}), y : \Pi w.(\{\nu : \mathbf{int} \mid \nu = w + 1\}\,\mathbf{ref}^{r_3}), z : \mathbf{int},$$
$$r_1 = 0 \le w \le 9 \Rightarrow 1, r_2 = 0 \le w \le 9 \Rightarrow 0.5 \text{ and } r_3 = -1 \le w \le 8 \Rightarrow 0.5\ .$$

Here, the refinement for y in the post-environment Γ_2 is *strengthened* by telling the type system that $y \boxplus 1$ is an alias of x.

Finally, the rule T-Assert requires the predicate φ to be valid under Γ.

Typing rules for function definitions and programs are the same as those in previous work [45, 47] and are given in the extended version.

Figure 7 describes how the **else** branch of `initMatrix` in Figure 1 is typed to express the intuition explained in Section 1.

- The pre-environment of the entire expression expresses that the pointer `pp` has full ownership to access the length-n outer array, whereas it has no ownership for the inner array (Line 1).
- The type environment at Line 3 expresses that, following T-MkIntArray, the pointer `s` has full ownership to access all the memory cells of the newly created length-n array; the refinement predicate of the type of `s` expresses that each memory cell is initialized by 0.
- Line 4 initializes `s` using the function `initArray`; we assume that `initArray` is typed $\langle \mathtt{l}:\mathbf{int}, \mathtt{n}:\mathbf{int}, \mathtt{p}:\Pi x_1.(\mathbf{int}\,\mathbf{ref}^{\,r_1})\rangle \to \langle \mathtt{l}:\mathbf{int}, \mathtt{n}:\mathbf{int}, \mathtt{p}:\Pi x_1.(\{\nu:\mathbf{int} \mid \varphi\}\,\mathbf{ref}^{\,r_2}) \mid \mathbf{int}\rangle$ elsewhere, where $r_1 = r_2 = 0 \le x_2 \le \mathtt{l}-1 \Rightarrow 1$ and $\varphi = 0 \le x_2 \le \mathtt{l}-1 \implies \nu = \mathtt{n}$, reflecting the intuition explained in Section 1. The type environment at Line 5 expresses that the ownership term in the type of `s` is unchanged by this function call and the refinement predicate expresses that each memory cell is initialized to n as required.
- The type environment at Lines 7–9 shows how T-Assign can be used to distribute the ownership retained by `s` between `pp` and `s`. In this example, all the ownership held by `s` is transferred to `pp`, resulting in the ownership term r_1 at Line 9 expressing that `pp` has the full ownership to access the inner array pointed to by the 0-th element of the outer array.
- At Line 10 where pointer arithmetic is conducted and T-AddPtr is applied, the ownership $0 \le x_2 \le \mathtt{n}-1 \Rightarrow 1$ held by `pp` for the outer array is split to $x_2 = 0 \Rightarrow 1$ and $1 \le x_2 \le \mathtt{n}-1 \Rightarrow 1$. The former is kept by `pp`. The latter is transferred to `t` bound to $\mathtt{pp} \boxplus 1$ after it is shifted by the offset 1 of this pointer arithmetic, resulting in $0 \le x_2 \le \mathtt{n}-2 \Rightarrow 1$, and hence the type of `t` at Line 13.
- Then, `initMatrix` is called recursively with the pointer to a matrix `t` at Line 14, changing the type of `t` so that it expresses that `t` retains full ownership for the outer array of length $\mathtt{n}-2$ and the inner array of length $\mathtt{n}-x_2-2$. Furthermore, the refinement predicate of the type of t at Line 15 expresses that the i-th inner integer array is initialized to the value $\mathtt{n}-i-1$ as required.
- Finally, the **alias** expression at Line 17 transfers back the ownership held by `t` to `pp`, resulting in the type of `pp` at Line 18.

Combined with the types of the other part of `initMatrix`, this function is typed as

$$\langle \mathtt{n}:\mathbf{int}, \mathtt{pp}:\Pi x_2.(\Pi x_1.(\mathbf{int}\,\mathbf{ref}^{\,\mathbf{0}})\,\mathbf{ref}^{\,r_1})\rangle \to \langle \mathtt{n}:\mathbf{int}, \mathtt{pp}:\Pi x_2.(\Pi x_1.(\{\nu:\mathbf{int} \mid \varphi\}\,\mathbf{ref}^{\,r_2})\,\mathbf{ref}^{\,r_3}) \mid \mathbf{int}\rangle$$

where $r_1 = r_3 = (0 \le x_2 \le \mathtt{n}-1 \Rightarrow 1, \mathbf{0})$ and $r_2 = (0 \le x_2 \le \mathtt{n}-1 \wedge 0 \le x_1 \le \mathtt{n}-x_2-1 \Rightarrow 1, \mathbf{0})$ and $\varphi = (0 \le x_2 \le \mathtt{n}-1 \wedge 0 \le x_1 \le \mathtt{n}-x_2-1 \implies \nu = \mathtt{n})$.

3.4 Soundness

We state a type soundness theorem below. We write $C \not\longrightarrow_D$ if there is no C' such that $C \longrightarrow_D C'$.

▶ **Theorem 3.1** (Soundness). *If $\vdash \langle D, e\rangle$ and $\langle \emptyset, \emptyset, e\rangle \longrightarrow_D^* C \not\longrightarrow_D$, then either (1) $C = \langle R, H, x\rangle$ for some R, H, and x, or (2) $C =$ **AliasFail**.*

The theorem above implies that an execution of a well-typed program does not get stuck, in particular, due to out-of-bounds access, pointer access through **null**, or assertion failures.

Remark on alias expressions and soundness. We emphasize that Theorem 3.1 does *not* assume the correctness of **alias** expressions. If an **alias** expression is wrong, the program reduces to **AliasFail**, which is an explicit, safely observable error state; not undefined

behavior. In particular, a well-typed program is guaranteed to be free from assertion failures, out-of-bounds accesses, and null-pointer dereferences, all of which are expressed as stuck states, regardless of whether the **alias** expressions hold at runtime.

From a practical standpoint, however, **alias** expressions do affect the *verifiability* of a program: an incorrect or missing **alias** expression may prevent the type checker from redistributing ownership appropriately, causing type inference to fail even for an otherwise correct program. As reported in Section 5 (Table 1), the majority of **alias** expressions required by our benchmarks are inserted automatically by our implementation; manual expressions are needed only in a small number of cases involving non-trivial pointer aliasing patterns. Notice that this design uses **alias** construct as an interface between the type system and any external alias analysis: The soundness proof of the type system (Theorem 3.1) does not depend on the particular alias analysis used to insert or discharge **alias** expressions. Our current implementation uses a simple syntactic automated **alias** insertion, but it can be replaced by more sophisticated alternatives without affecting the metatheory.[2]

4 Type Inference

We describe the type inference procedure used in the experiments reported in Section 5. The type inference procedure consists of the following three steps:

- Step 1: Simple type inference inferring the simple type of each variable.
- Step 2: Ownership inference inferring ownership expressions for each reference type.
- Step 3: Refinement inference inferring refinement predicates for each refinement type.

Step 1 is the standard unification-based type inference.

The main strategy of Steps 2 and 3 is to generate constraints based on the syntax-directed typing rules derived from the rules introduced in Section 3.3. To make type inference tractable, we use a template-based approach for ownership inference (Step 2). Using the obtained ownership expressions, we then generate CHC constraints for refinement inference (Step 3). We explain Steps 2 and 3 in detail in the following subsections.

4.1 Step 2: Ownership Inference

Step 2.1: Generating Ownership Constraints

Step 2 of our type inference procedure is based on a template-based approach: We prepare a template for each ownership expression associated with a reference type, generate a set of constraints based on the typing rules, and solve the constraints to obtain a concrete ownership expression as a solution. Concretely, we prepare a template type environment at each location in a simply-typed program, in which each reference type is accompanied by a template ownership expression.

In the present implementation, a simple type $\tau^-\ \mathbf{ref}$ is promoted to $\Pi x.(\tau\ \mathbf{ref}^r)$, where τ is obtained by promoting τ^- recursively and choosing the template for r from one of (1) $x \in [l, h] \Rightarrow o$, (2) $x = 0 \Rightarrow o_1, x \in [1, h] \Rightarrow o_2$ and (3) $y = 0 \wedge x \in [l_1, h_1] \Rightarrow o_1, y \in [1, h_2] \wedge x \in [l_3, h_3] \Rightarrow o_2$. Here, o, o_1, and o_2 are variables representing rational numbers; l and h are linear combinations of the form $c_0 + c_1x_1 + \cdots + c_{k-1}x_k + c_kx$, where c_i are variables representing integers; $x, x_1, \ldots, x_{k-1}$ are variables of integer type available at this program point. We explain the variable y in Template (3) later.

[2] The detail of our **alias** insertion algorithm and the alias analysis it relies on are described in the extended version.

Template (2) expresses that a pointer to an array has ownership o_1 for the head element, ownership o_2 for the other elements in an interval of indices, and 0 for all the other elements. Template (1) is a special case of (2) where the ownership of the head element is the same as the other elements in the array. These templates are useful in expressing an access pattern of a pointer that accesses the head element differently from the other elements; this pattern is frequently used in a program that iterates over an array via a pointer. Although Template (2) is more expressive than (1), we found that using both (1) and (2) enhances the performance of the type inference procedure.

For nested reference types (e.g., $\Pi x_2.(\Pi x_1.\mathbf{int}\,\mathbf{ref}^{r_1})\,\mathbf{ref}^{r_2}$), where the outermost ownership expression (e.g., r_2) is represented using Template (2), the type inference procedure uses Template (3) for the inner ownership expression (e.g., r_1), with y being the index variable of the outermost array type (e.g., x_2). This template enables another access pattern of a program that iterates over a nested array where its 0-th row is processed differently from the other rows.

Then, the type inference procedure generates constraints based on the typing rules in a forward-reasoning style: Given an expression e and a pre-environment Γ, the procedure generates a set of constraints C and a post-environment Γ', potentially involving recursive calls to the procedure if e is a compound expression. At the beginning of a program or function and for a recursive call, the type inference procedure prepares a pre-environment; the templates for the ownership terms in this pre-environment are chosen from Templates (1–3) based on the following heuristics, which are designed so that Templates (2) and (3) are used only where the 0-th element of an array needs to be treated specially.

- Template (1) is used at the beginning of a program or function or on array creation, where we do not need special treatment of the head element of an array template. The type of x in the pre-environment of e in $\mathbf{let}\,y = *x\,\mathbf{in}\,e$ and $x := y\,;e$ uses Template (2) or (3) since the ownership for the head element of x often differs from that of the other elements. The type of y in the pre-environment of e uses Template (1).
- For a pointer-arithmetic expression ($\mathbf{let}\,x = y \boxplus z\,\mathbf{in}\,e$), the type of y in the pre-environment of e uses the same template as that in the pre-environment of this expression. The type of x in the pre-environment of e uses Template (1).
- If the type of y in the pre-environment Γ_1 for $\mathbf{alias}(x = *y)\,;e$ uses Template (1), we prepare a type environment Γ_2 in which the type of y is expressed using Template (2) or (3) and the rest is the same as Γ_1, generate a subtyping constraint $\Gamma_1 \leq \Gamma_2$, and use Γ_2 as the pre-environment for e.
- If the type of x in the pre-environment of $\mathbf{alias}(x = y \boxplus z)\,;e$ uses Template (2) or (3), we prepare a pre-environment in which it uses Template (1) using a subtyping constraint and use it as the pre-environment for this expression in the same way as described in the case of $\mathbf{alias}(x = *y)\,;e$.
- At the end of a function body and at the end of each branch of an **if** expression, all reference types are forced to use Templates (2) and (3) using subtyping constraints in the same way as described in the case of $\mathbf{alias}(x = *y)\,;e$.

Although the above heuristics are not complete in that there exist well-typed programs that cannot be typed under these restrictions, we will demonstrate in Section 5 that these heuristics successfully handle the benchmarks used in our experiments.

We remark that the implementation by Tanaka et al. [45] uses only Template (1). As demonstrated in experimental results in Section 5, we need to use Templates (2) and (3) to handle frequent access patterns in matrix-manipulating programs.

```
1  // pp:Πx2.(Πx1.(int ref^{x1∈[l^pp_{0,0},h^pp_{0,0}] ⇒ o^pp_{0,0}}) ref^{x2∈[l^pp_{0,1},h^pp_{0,1}] ⇒ o^pp_{0,1}})
2  // s:Πx1.(int ref^{x1∈[l^s_{0,0},h^s_{0,0}] ⇒ o^s_{0,0}})
3  pp := s;
4  // pp:Πx2.(Πx1.(int ref^{r1,1}) ref^{r1,2})
5  // r1,1 = (x2 = 0) ∧ x1 ∈ [l^pp_{1,0}, h^pp_{1,0}] ⇒ o^pp_{1,0}, (1 ≤ x2 ≤ h^pp_{1,3}) ∧ x1 ∈ [l^pp_{1,1}, h^pp_{1,1}] ⇒ o^pp_{1,1}
6  // r1,2 = (x2 = 0) ⇒ o^pp_{1,2}, x2 ∈ [1, h^pp_{1,3}] ⇒ o^pp_{1,3}
7  // s:Πx1.(int ref^{x1∈[l^s_{1,0},h^s_{1,0}] ⇒ o^s_{1,0}})
8  let t = pp ⊞ 1 in
9  // pp:Πx2.(Πx1.(int ref^{r2,1}) ref^{r2,2})
10 // r2,1 = (x2 = 0) ∧ x1 ∈ [l^pp_{2,0}, h^pp_{2,0}] ⇒ o^pp_{2,0}, (1 ≤ x2 ≤ h^pp_{2,3}) ∧ x1 ∈ [l^pp_{2,1}, h^pp_{2,1}] ⇒ o^pp_{2,1}
11 // r2,2 = (x2 = 0) ⇒ o^pp_{2,2}, x2 ∈ [1, h^pp_{2,3}] ⇒ o^pp_{2,3}
12 // t:Πx2.(Πx1.(int ref^{r'2,1}) ref^{r'2,2})
13 // r'2,1 = x1 ∈ [l^t_{2,0}, h^t_{2,0}] ⇒ o^t_{2,0},
14 // r'2,2 = x2 ∈ [l^t_{2,1}, h^t_{2,1}] ⇒ o^t_{2,1}
```

Figure 8 Code snippet with ownership expression templates.

The generated constraints are solved using an SMT solver to determine the values for x_i and c_i. These values are substituted into the templates to obtain concrete ownership terms associated with the types at each location.

For example, consider the following code snippet, where the simple types for pp and s are **int ref** and **int**, respectively.

$$pp := s; \mathbf{let}\ t = pp \boxplus 1\ \mathbf{in} \ldots$$

The type inference algorithm prepares a template for each program location in Figure 8, wherein the type templates are shown in green.

Because an assignment to the array occurs, the template of pp after assignment treats the head element separately. Then, the following constraints are generated from the first statement of the snippet and T-Assign:

$$o^{pp}_{0,1} = 1 \land \forall n > 0. l^{pp}_{0,1} \leq 0 \leq h^{pp}_{0,1}$$
$$\forall n > 0. (l^{pp}_{0,1} \leq 0 \land \max\{h^{pp}_{1,3}, 0\} \leq h^{pp}_{0,1} \land o^{pp}_{1,2} \geq o^{pp}_{0,1} \land o^{pp}_{1,3} \geq o^{pp}_{0,1})$$
$$\forall n > 0. (\forall x_2. x_2 = 0 \Rightarrow (l^{s}_{0,0} \leq l^{pp}_{1,0} \land h^{pp}_{1,0} \leq h^{s}_{0,0} \land o^{pp}_{1,0} \leq o^{s}_{0,0} \land o^{s}_{1,0} = 0))$$
$$\forall n > 0. (\forall x_2. ((1 \leq x_2 \leq h^{pp}_{1,3}) \Rightarrow (l^{pp}_{0,0} \leq l^{pp}_{1,1} \land h^{pp}_{1,1} \leq h^{pp}_{0,0} \land o^{pp}_{1,1} \leq o^{pp}_{0,0}))).$$

We remark that, for each nested reference type, we generate *Empty* constraints that we mentioned in Section 3.1. For example, for each occurrence of $\Pi x_2.(\Pi x_1.\mathbf{int\ ref}^{x_1 \in [l_1, h_1] \Rightarrow o_1})\ \mathbf{ref}^{x_2 \in [l_2, h_2] \Rightarrow o_2}$, the following constraints are generated: $(o_2 = 0 \Rightarrow o_1 = 0) \land (x_2 < l_2 \Rightarrow o_1 = 0 \lor h_1 < l_1) \land (x_2 > h_2 \Rightarrow o_1 = 0 \lor h_1 < l_1)$.

Step 2.2: Solving Ownership Constraints

Then, the type inference procedure solves the constraints gathered from the entire program to find a solution for unknowns. The generated constraints are in the form of $\exists \vec{c}, \vec{d}, \vec{o}. \forall \vec{x}. \varphi$, where $\vec{c}, \vec{d}, \vec{o}$ are unknowns representing the coefficients and ownership values in the templates and $\vec{x}$ are program variables and array indices. The same form of constraints is also generated by the type-inference procedure by Tanaka et al. To solve these $\exists\forall$ constraints using an SMT solver, Tanaka et al. use the following "guess-and-check" method. Specifically, given a constraint of the form $\exists \vec{c}. \forall \vec{x}. \psi(\vec{c}, \vec{x})$, they first generate a set of random integers $\{\vec{n}_1, \ldots, \vec{n}_k\}$ and weaken the constraint to $\psi(\vec{c}, \vec{n}_1) \land \cdots \land \psi(\vec{c}, \vec{n}_k)$. The satisfiability of this weakened

```
1  // pp:Πx2.(Πx1.({ν : int | P0(n, x1, x2, ν)} ref^0) ref^{x2∈[0,n−1] ⇒ 1})
2  // s:Πx1.({ν : int | P1(n, x1, ν)} ref^{x1∈[0,n−1] ⇒ 1})
3  pp := s;
4  // pp:Πx2.(Πx1.({ν : int | P2(n, x1, x2, ν)} ref^{r1,1}) ref^{r1,2})
5  // r1,1 = (x2 = 0) ∧ x1 ∈ [0, n − 1] ⇒ 1, (1 ≤ x2 ≤ n − 1) ∧ x1 ∈ [0, 0] ⇒ 0
6  // r1,2 = (x2 = 0) ⇒ 1, x2 ∈ [1, n − 1] ⇒ 1
7  // s:Πx1.({ν : int | P3(n, x1, ν)} ref^{x1∈[0,0] ⇒ 0})
8  let t = pp ⊞ 1 in
9  // pp:Πx2.(Πx1.({ν : int | P4(n, x1, x2, ν)} ref^{r2,1}) ref^{r2,2})
10 // r2,1 = (x2 = 0) ∧ x1 ∈ [0, n − 1] ⇒ 1, (1 ≤ x2 ≤ 0) ∧ x1 ∈ [0, 0] ⇒ 0
11 // r2,2 = (x2 = 0) ⇒ 1, x2 ∈ [1, 0] ⇒ 0
12 // t:Πx2.(Πx1.({ν : int | P5(n, x1, x2, ν)} ref^0) ref^{x2∈[0,n−2] ⇒ 1})
```

Figure 9 Code snippet with refinement type templates.

constraint is then checked using an SMT solver to obtain a candidate solution $\vec{m}$ for $\vec{c}$; if it is unsatisfiable, then the original constraint is unsatisfiable. Then, again using an SMT solver, the validity of $\forall \vec{x}.\psi(\vec{m}, \vec{x})$ is checked; if it is valid, then $\vec{m}$ is accepted as the solution, and the ownership inference terminates. Otherwise, the number of random samples k is increased, and the process is repeated until a correct $\vec{m}$ is found or the problem is found to be unsatisfiable.

Although the overall structure of our procedure is similar to that of Tanaka et al., we improve their procedure as follows. If a solution $\vec{m}$ for $\vec{c}$ in $\psi(\vec{c}, \vec{n}_1) \wedge \cdots \wedge \psi(\vec{c}, \vec{n}_k)$ is obtained but $\forall \vec{x}.\psi(\vec{m}, \vec{x})$ is not valid, then instead of choosing the next value $\vec{n}_{k+1}$ randomly as they do, we use the obtained counterexample for $\forall \vec{x}.\psi(\vec{m}, \vec{x})$ as $\vec{n}_{k+1}$; so $\vec{n}_{k+1}$ satisfies $\neg\psi(\vec{m}, \vec{n}_{k+1})$. By this improvement, we can guarantee progress of the procedure since the new search space $\{\vec{m} \mid \psi(\vec{m}, \tilde{n}_1) \wedge \ldots \wedge \psi(\vec{m}, \tilde{n}_k) \wedge \psi(\vec{m}, \tilde{n}_{k+1})\}$ is strictly smaller than the previous search space $\{\vec{m} \mid \psi(\vec{m}, \tilde{n}_1) \wedge \ldots \wedge \psi(\vec{m}, \tilde{n}_k)\}$.

4.2 Step 3: Refinement Inference

Once ownership expressions for each reference type are determined, the type inference procedure conducts refinement inference in a manner similar to that of Tanaka et al. Our procedure extends each simple integer type **int** with a predicate variable P that represents a predicate over integers to create an integer type template $\{\nu : \mathbf{int} \mid P\}$, generates constraints over the predicate variables as constrained Horn clauses (CHC), and solves the constraints using a CHC solver. For each predicate variable P in an integer type $\{\nu : \mathbf{int} \mid P\}$, P may refer to the variables $\vec{x}$ consisting of all the integer variables in the type environment at the program point and ν.

For an example of refinement type inference, we use the same code snippet as in the ownership inference section. Figure 9 shows the type environments with refinement type templates associated at each program location in the code snippet. For example, from $pp := s$ and T-Assign, the following CHC is generated.

$$\forall n, x_1, x_2, \nu.n > 0 \wedge x_2 = 0 \wedge P_1(n, x_1, \nu) \Rightarrow P_2(n, x_1, x_2, \nu)$$
$$\forall n, x_1, x_2, \nu.n > 0 \wedge 0 < x_2 \leq n - 1 \wedge P_0(n, x_1, x_2, \nu) \Rightarrow P_2(n, x_1, x_2, \nu)$$

This constraint is solved using a CHC solver to obtain solutions for P_1 and P_2; these are substituted back into the original type templates to obtain a type derivation of the given program.

4.3 Properties of the Type-Inference Procedure

This section discusses the properties of our type-inference procedure. Since the procedure itself is similar to the one by Tanaka et al., we discuss the properties in prose rather than mathematical formalisms.

Soundness. The ownership inference step of the type-inference procedure generates constraints whose satisfiability implies that the given program is well-typed under our type system, assuming that each ownership expression is expressible using the templates. Therefore, if a solution to the generated constraints is found, then the program is well-typed under our type system.

The refinement inference step follows the standard approach of reducing refinement constraints to constrained Horn clauses (CHCs) and solving them with a CHC solver, as adopted in previous refinement-type-inference procedures [28, 3, 16]. This step is sound provided that the employed CHC solver is sound, meaning that every model it returns is a correct solution to the CHCs. This style of argument – reducing soundness of refinement-type inference to the soundness of the underlying constraint solver – is standard in refinement-type systems; see, e.g., Rondon [38, Appendix A].

Completeness. In contrast, neither Step 2 nor Step 3 of the type-inference procedure is complete. The ownership inference step is based on a fixed finite set of templates for ownership terms. As is standard in template-based program verification, this immediately implies incompleteness: the algorithm may fail to find ownership terms that lie outside the chosen template language, even when the program is well-typed. Similarly, the refinement inference step may fail to find a solution since CHC solving is not complete in general [12].

Complexity. It is difficult to characterize the time complexity of the entire procedure since our type-inference procedure invokes Z3 to solve ownership and CHC constraints. In the following, we show an upper bound on the size of the generated constraints. Let m be the size of the input program and n be the maximum nesting depth of the **ref** type constructor appearing in the input program. Each application of a syntax-directed typing rule generates ownership constraints and CHC constraints of size $O(n)$. Since the total number of syntax-directed rule applications is $O(m)$, the overall number of generated constraints is $O(nm)$.

5 Experiments

Some benchmark details and supplementary experimental results are deferred to the extended version of this paper.

We implemented the type inference procedure described in Section 4 and evaluated it using various benchmarks. We aim to address the following research questions through our experiments:

RQ1 Can our verifier effectively analyze programs containing nested arrays, a feature not supported by Tanaka et al.'s verifier, and produce useful verification results?

RQ2 Can our verifier handle the same set of programs verifiable by Tanaka et al.'s implementation without significant overhead?

All the experiments were conducted on a machine with an Apple M2 CPU and 24 GB of RAM. Our implementation is written in OCaml 4.14.1; we used Z3 4.14.1 [13] as an SMT solver and HoIce 1.10.0 [8] as a CHC solver.

There is a gap between our implementation and the formal theory. Our implementation simplifies ownership-template selection during type inference: for example, certain pointer-arithmetic forms force a conversion from Template (2) to Template (1), and fixed template assumptions are imposed at **alias** expressions, function boundaries, and branch endpoints to accelerate inference. The implementation also extends the input language with several conveniences not present in the formal calculus, including explicit ownership annotations on function signatures; nondeterministically chosen integer literals (_) with optional refinement constraints; refinement-annotated array allocations; a distinction between ownership-partitioning and ownership-sharing modes of pointer arithmetic; restricted ownership-distribution rules for reads, writes, and alias expressions; and extended **assert** syntax for accessing nested-array elements.

Our implementation also conducts automated insertion of **alias** expressions at program points where an **alias** expression is guaranteed to be correct and ownership redistribution is possible, to reduce the annotation burden on the programmer. Concretely, our verifier inserts the **alias** expressions according to the following rules:

- In e of $\mathbf{let}\, x = *y\, \mathbf{in}\, e$, if there is an expression that creates a new binding through y (i.e., an expression of the form $\mathbf{let}\, z = *y\, \mathbf{in}\, e_1$ or $\mathbf{let}\, z = y \boxplus n\, \mathbf{in}\, e_1$), then $\mathbf{alias}(x = *y)$ is automatically inserted immediately before that expression. By this insertion, a part of the ownership of x as well as that of y can be transferred to the new binding z.
- In e of $\mathbf{let}\, x = *y\, \mathbf{in}\, e$, if no such expression exists, or if the alias was instead created via a pointer arithmetic expression $\mathbf{let}\, x = y \boxplus n\, \mathbf{in}\, e$, then the corresponding **alias** expression is inserted at the end of e (i.e., at the point where the scope of x ends).

We designed the above rules to cover common patterns in which ownership redistribution is required and inserted **alias** expression is correct. The previous work by Tanaka et al. [45] also implements a similar automated insertion of **alias** expressions for non-nested arrays.

These simplifications and extensions are designed to make verification tractable while retaining sufficient verification power, as demonstrated by the experiments in Section 5. The full details of each item are given in the full version.

5.1 RQ1: Verification of Programs with Nested Arrays

5.1.1 Benchmarks

To answer RQ1, we wrote programs that manipulate matrices – one of the main use cases of nested arrays in practice – in our target language and verified them using our verifier. The detailed explanation of each benchmark program is given in the full version. The source code is included in the supplementary material. We only describe the rationale behind the design of our benchmark suite here.

The benchmark suite was designed to systematically cover typical programming patterns that arise when manipulating matrix-like data structures with pointer arithmetic and aliasing. The benchmarks vary along three main dimensions: (i) array shape (rectangular vs. non-rectangular), (ii) traversal order (row-major, column-major, and diagonal-like traversals), and (iii) the number of matrices and aliasing/ownership patterns involved (single-matrix updates vs. coordinated updates of multiple matrices with shared read access).

Concretely, several benchmarks operate on *rectangular* matrices, where every row has the same length; typical examples include programs that initialize or update all elements of a dense matrix (e.g., Init-Matrix, Sum-Matrix, Add-Matrix, Trans-Matrix), whose ownership terms do not need to distinguish rows by length. In contrast, other benchmarks use *non-rectangular* shapes such as lower-triangular matrices (e.g., Indexed-Matrix); in these

Table 1 The results of the experiments for RQ1. Times are presented in seconds. The rightmost three columns show the number of **alias** expressions in each benchmark program, broken down into those inserted manually by the programmer and those inserted automatically by the verifier.

Name	Total time (ownership/refinement) Ours	# of alias Manual	Auto	Total
Init-Matrix(2D)	5.093 (2.412+2.680)	0	2	2
Init-Matrix(3D)	205.544 (16.437+189.106)	0	3	3
Indexed-Matrix(2D)	36.895 (0.433+36.462)	0	2	2
Indexed-Matrix(3D)	17.232 (13.311+3.920)	0	3	3
Indexed-Matrix(4D)	350.784 (114.746+236.037)	0	4	4
Indexed-Value	3.336 (2.943+0.393)	0	2	2
Sum-Matrix	48.244 (31.694+16.549)	0	5	5
Copy-Matrix	131.380 (80.184+51.195)	0	7	7
Add-Matrix	208.473 (191.750+16.722)	0	10	10
Trace-Matrix	54.533 (10.845+43.688)	1	4	5
Trans-Matrix	163.251 (79.337+83.913)	1	4	5
Eta-Equ-Sum	19.318 (17.646+1.671)	0	6	6
Eta-Equ-Trace	18.331 (16.563+1.767)	3	3	6
Lower-Triangle	26.132 (3.053+23.079)	0	2	2
Swap	75.411 (73.636+1.774)	4	4	8
Compare-Element	16.550 (2.110+14.439)	0	2	2
Row-Add	28.302 (4.563+23.739)	0	4	4
Boomerang	17.674 (14.742+2.931)	0	4	4
Share-Add-Matrix	376.867 (29.328+347.538)	3	5	8

programs, the length of each inner array depends on the outer index, so the ownership terms for inner arrays must depend on outer indices, directly stressing the main generalization of our system over Tanaka et al.'s work.

The suite also covers a range of access and aliasing patterns. Row-major traversal appears in initialization and accumulation programs (such as Init-Matrix, Indexed-Value, Sum-Matrix, Copy-Matrix, Row-Add, Share-Add-Matrix), which exercise ownership templates tailored to pointer iteration where the head element is treated differently from the remaining elements. Column-major traversal is represented by transpose-like programs (e.g., Trans-Matrix), where the access pattern repeatedly visits the same column across different rows, while diagonal-style traversals (e.g., Trace-Matrix and variants) visit entries whose indices satisfy constraints such as $i = j$ or $i \leq j$.

The benchmarks also differ in the number of matrices manipulated simultaneously: some programs manipulate a single matrix in place (Init-Matrix, Indexed-Matrix, Lower-Triangle, Boomerang), whereas others operate on two or more matrices (Copy-Matrix, Add-Matrix, Compare-Element), requiring the system to track ownership of multiple nested arrays with correlated shapes.

Finally, examples such as Share-Add-Matrix use explicit aliasing of pointers to express shared read access to source matrices while accumulating results into a separate target matrix. This benchmark stresses the interaction between **alias** expressions, ownership splitting/merging, and nested-array shapes. Furthermore, the suite is intended to cover a representative range of nested-array usage patterns in low-level matrix code, and none of these nested-array benchmarks can be verified by the previous verifier of Tanaka et al.

Table 2 Inferred ownership terms for the matrix pointer after initialization / main computation. x_1 and x_2 denote the inner and outer array indices, respectively; x_3 denotes the outermost index in 3D examples; n, k, l are function parameters in scope. T1/T2/T3 indicate which type of template introduced in Section 4.1 was used.

Program	r_{outer}	r_{inner}
INDEXED-MATRIX(2D)	$0 \le x_2 \le n-1 \mapsto 1$ (T1)	$0 \le x_2 \le n-1 \wedge 0 \le x_1 \le n-x_2-1 \mapsto 1$ (T3)
Inner ownership depends on the outer index x_2.		
SUM-MATRIX	$x_2 = 0 \mapsto 1$, $1 \le x_2 \le n_2-1 \mapsto 1$ (T2)	$0 \le x_1 \le n_1-1 \mapsto 1$ (T1)
Outer ownership uses T2: head row is dereferenced before advancing the pointer.		
TRANS-MATRIX	$0 \le x_2 \le n_2-1 \mapsto 1$ (T1)	$0 \le x_1 \le n_1-1 \mapsto 1$ (T1)
Uniform rectangular matrix; column-major traversal.		
SHARE-ADD-MATRIX	$0 \le x_2 \le n_2-1 \mapsto 0.1$ (T1)	$0 \le x_1 \le n_1-1 \mapsto 0.1$ (T1)
Fractional (read-only) sharing.		
	r_{outer}	r_{mid} / r_{inner}
INDEXED-MATRIX (3D)	$0 \le x_3 \le m-1 \mapsto 1$ (T1)	$0 \le x_2 \le m-x_3-1 \mapsto 1$ (T3) $0 \le x_1 \le m-x_3-x_2-1 \mapsto 1$ (T3)
Inner ownership depends on x_3; innermost depends on both x_3 and x_2.		

5.1.2 Results

Table 1 presents the time spent on verifying each program with our implementation. In addition to the total verification time, we also provide a breakdown of time spent on ownership inference and refinement type verification. Depending on the benchmark, the total verification time varies from around 5 seconds to about 6 minutes; none exceeded the time or memory limits, despite the modest hardware used.

To illustrate the variety of ownership terms used by the benchmarks, we present the ownership terms inferred for the post-type of the main matrix pointer in representative programs. Table 2 summarizes the results; we write r_{outer} for the ownership term associated with the outermost **ref** type and r_{inner} for the one associated with the next inner **ref** type (and r_{mid} for the middle level in 3D examples).

Several observations are worth noting. First, all three templates are exercised across the benchmark suite. Template 1 suffices when rows have uniform length and uniform access permissions, as in TRANS-MATRIX. Template 2 is required whenever pointer iteration treats the head element differently from the tail, which arises in every program that traverses a matrix via recursive pointer arithmetic (e.g., INIT-MATRIX, SUM-MATRIX, COPY-MATRIX). Template 3, which introduces a dependency on an outer index variable, is required for non-rectangular matrices and for the inner ownership terms in all programs where the length of an inner array varies with the outer index (e.g., INDEXED-MATRIX).

Second, several benchmarks require ownership terms that depend on variables other than array indices. For example, Figure 10 shows how the 3D version of INDEXED-MATRIX is typed. The ownership term for the innermost array after the recursive call to `iTm` is inferred as $0 \le x_1 \le m - x_3 - x_2 - 1 \mapsto 1$, where m is the function parameter. This demonstrates that the generality of our ownership terms, allowing dependency on outer indices *and* on variables in the type environment, is essential for verifying programs with higher-dimensional nested arrays.

Third, the SHARE-ADD-MATRIX benchmark demonstrates a qualitatively different use of ownership terms: the inferred ownership is 0.1 (read-only) rather than 1 (read-write), reflecting that two aliases to the same matrix share read access while a third matrix receives full write access. This pattern exercises the fractional ownership aspect of our system jointly with nested-array reasoning.

```
1  iTm(m, ppp)
2  // ppp:Πx3.(Πx2.(Πx1.({ν:int | ⊤} ref^0) ref^0) ref^{x3∈[0,m−1] ⇒ 1})
3  {
4    if m > 0 then {
5      let r = alloc m : int ref ref in
6  // r:(Πx2.(Πx1.({ν:int | ⊤} ref^0) ref^{x2∈[0,m−1] ⇒ 1}))
7      let d = iM(m, r) in
8  // r:(Πx2.(Πx1.({ν:int | 0 ≤ x2 ≤ m − 1 ∧ 0 ≤ x1 ≤ m − x2 − 1 ⟹ ν = 1} ref^{r1}) ref^{r2}))
9  // r1 = x1 ∈ [0, m − x2 − 1] ⇒ 1, r2 = (x2 ∈ [0, m − 1] ⇒ 1)
10     ppp := r;
11 // ppp:(Πx3.(Πx2.(Πx1.({ν:int | x3 = 0 ∧ 0 ≤ x2 ≤ m − 1 ∧ 0 ≤ x1 ≤ m − x2 − 1 ⟹ ν =
   1} ref^{r1}) ref^{r2}) ref^{r3}))
12 // r1 = (x3 = 0 ∧ x1 ∈ [0, m − x2 − 1] ⇒ 1), r2 = (x3 = 0 ∧ x2 ∈ [0, m − 1])
13 // r3 = (x3 ∈ [0, m − 1] ⇒ 1)
14 // r:(Πx2.(Πx1.({ν:int | ⊤} ref^0) ref^0))
15     let q = ppp ⊞ 1 in
16 // ppp:(Πx3.(Πx2.(Πx1.{ν:int | x3 = 0 ∧ 0 ≤ x2 ≤ m − 1 ∧ 0 ≤ x1 ≤ m − x2 − 1 ⟹ ν =
   1}) ref^{r1}) ref^{r2}) ref^{r3}
17 // r1 = (x1 ∈ [0, m − x2 − 1] ⇒ 1), r2 = (x2 ∈ [0, m − 1]), r3 = (x3 = 0 ⇒ 1)
18 // q:(Πx3.(Πx2.(Πx1.({ν:int | ⊤} ref^0) ref^0) ref^{x3∈[0,m−2] ⇒ 1}))
19     let d2 = iTm(m-1, q) in
20 // q:(Πx3.(Πx2.(Πx1.({ν:int | 0 ≤ x3 ≤ m−2∧0 ≤ x2 ≤ m−x3−2∧0 ≤ x1 ≤ m−x3−x2−2 ⟹
   ν = 1} ref^{r1}) ref^{r2}) ref^{r3}))
21 // r1 = (x1 ∈ [0, m − x3 − x2 − 2] ⇒ 1), r2 = (x2 ∈ [0, m − x3 − 2] ⇒ 1), r3 = (x3 ∈ [0, m − 2] ⇒ 1)
22     alias(q=ppp+1);0
23 // ppp:(Πx3.(Πx2.(Πx1.({ν:int | 0 ≤ x3 ≤ m−1∧0 ≤ x2 ≤ m−x3−1∧0 ≤ x1 ≤ m−x3−x2−1 ⟹
   ν = 1} ref^{r1}) ref^{r2}) ref^{r3}))
24 // r1 = (x1 ∈ [0, m − x3 − x2 − 1] ⇒ 1), r2 = (x2 ∈ [0, m − x3 − 1] ⇒ 1), r3 = (x3 ∈ [0, m − 1] ⇒ 1)
25 // q:(Πx3.(Πx2.(Πx1.({ν:int | ⊤} ref^0) ref^0) ref^0))
26   } else { 0 }
27 }
```

Figure 10 Typing example of IndexedMatrix (3D).

We also evaluated the effectiveness of our automated **alias** insertion mechanism. The rightmost three columns of Table 1 show the number of **alias** expressions required in each benchmark program from Table 1, distinguishing between those that were manually written by the programmer and those that were automatically inserted by the verifier using the mechanism described above. In 14 out of 19 benchmarks, no manual insertion of **alias** expressions was needed at all; even in the remaining benchmarks, the number of manually inserted **alias** expressions is at most 4. These results demonstrate that the automated insertion mechanism covers the majority of **alias** expressions required in practice, substantially reducing the annotation burden on the programmer.

Based on these results, our conclusion to RQ1 is affirmative: Our verifier effectively analyzes programs containing nested arrays, a feature not supported by Tanaka et al.'s verifier, and produces useful verification results.

5.2 RQ2: Comparison with Prior Work

To address RQ2, we compared our verifier with the implementation by Tanaka et al. [45] using the same benchmark suite they employed. The detailed explanation of each benchmark program is given in the full version. The source code is included in the supplementary material.

Table 3 shows the results of the experiments comparing our verifier with that of Tanaka et al. Since the implementation by Tanaka et al. was tested with an older version of Z3 (4.11.2), we conducted our experiments using two different Z3 versions. The verifier by Tanaka et al.

Table 3 The results of the experiments for RQ2. Times are presented in seconds. The fastest time for each benchmark is shown in bold.

	Total time(ownership/refinement)			
	Tanaka+		**Ours**	
z3 version	4.11.2	4.14.1	4.11.2	4.14.1
Init-10	**0.255 (0.052+0.202)**	0.262 (0.041+0.220)	0.590 (0.193+0.397)	0.493 (0.102+0.390)
Init-1000	3.689 (3.466+0.223)	157.486 (0.299+157.186)	**1.102 (0.684+0.417)**	2.274 (0.332+1.942)
Sum	1.962 (1.797+0.165)	**0.266 (0.138+0.128)**	1.056 (0.202+0.854)	0.337 (0.110+0.227)
Sum-Back	0.471 (0.134+0.337)	0.836 (0.509+0.326)	0.335 (0.224+0.111)	**0.224 (0.119+0.105)**
Sum-Both	1.078 (0.575+0.503)	**0.739 (0.310+0.428)**	0.916 (0.702+0.214)	0.919 (0.182+0.736)
Sum-Div	0.789 (0.361+0.427)	0.780 (0.341+0.439)	0.777 (0.630+0.146)	**0.647 (0.492+0.155)**
Copy-Array	30.585 (28.648+1.937)	61.470 (61.203+0.266)	1.180 (0.622+0.558)	**1.036 (0.318+0.717)**
Add-Array	328.066 (327.652+0.413)	timeout($> 10^3$)	**38.205 (2.848+35.357)**	338.428 (1.315+337.112)

fails to verify **Add-Array** due to a timeout when using the latest version of Z3, so we used an older version (4.11.2) for that benchmark. We can observe that our verifier successfully verifies all the benchmarks that Tanaka et al.'s verifier can handle. Regarding the verification time for each benchmark, we observe that our verifier incurs considerably less overhead overall than Tanaka et al.'s verifier. For **Copy-Array** and **Add-Array**, ownership inference by our verifier is significantly faster than by Tanaka et al.'s verifier. As is evident from the experimental results, the time required for ownership inference is highly dependent on the version of Z3. Furthermore, although we did not change the HoIce version in this experiment, the refinement type inference for Init-1000 from Tanaka et al. is also significantly affected by the version of Z3. We conjecture that differences in the Z3 version affect ownership-inference results, and it is likely that the resulting refinement constraints were incompatible with the CHC solver (HoIce), causing the significant time increase. We conducted an additional experiment to evaluate the dependency of the verification time on the Z3 version for our verifier in the full version.

Based on these results, our conclusion to RQ2 is affirmative: Our verifier can handle the same set of programs that are verifiable by the implementation from the prior work by Tanaka et al. without much overhead.

6 Discussion

Applicability to mainstream languages. Our target language is a C-like imperative language with explicit pointer arithmetic and heap allocation. Applying our type system is most natural for *type-safe subsets* of C, programs that do not perform arbitrary pointer casts; many numerical and scientific programs – in which nested arrays are heavily used – already satisfy these restrictions. The **alias** construct can be seen as a dynamically checked annotation that replaces the need for a separate must-alias analysis. Our method can be used to verify functional correctness properties for such programs.

Beyond C, our system has a natural connection to modern typed languages. For example, in OCaml, mutable arrays and references are first-class values (although it does not support pointer arithmetic). Our method could be used to verify functional correctness of OCaml programs in which mutable arrays and references are involved by accommodating ownership.

In this regard, Cameleer [35], a functional-correctness verifier for OCaml, already supports mutable references [35]; however, their backend Why3 poses a restriction that all the aliases of a reference and an array must be known statically. Our ownership-based approach could provide a more flexible alternative to Cameleer's approach.

For Rust, the borrow checker already enforces a strict ownership discipline at the language level, and mature verification tools such as Prusti [1] and Creusot [14] leverage this discipline for functional correctness verification. Our work addresses a different setting, languages without built-in ownership guarantees, where aliasing must be tracked explicitly. Nevertheless, the idea of index-dependent ownership terms could complement Rust verification tools when reasoning about unsafe code blocks that perform raw pointer arithmetic on nested arrays, a direction we leave for future work.

Towards more general heap structures. Although the present work focuses on nested arrays, our formulation of ownership terms can be applied to more general data structures. Ownership terms are, in essence, predicate-guarded maps from indices to fractional permissions, and this mechanism is not inherently tied to arrays. For example, pairs and tuples can be handled by extending ownership expressions to allow mentioning the position of each component. C-like structures also can be handled by extending ownership expressions to handle the fields of a structure. However, further extensions for heap-allocated data structures such as linked lists and trees would be more challenging, as they require introducing recursive types and suitably extended ownership terms.

7 Related Work

Toman et al. [47] have proposed a type system and its implementation for verifying functional correctness of imperative programs that support mutable references and aliasing. Their type system is based on refinement types and fractional ownership types, which provide flow-sensitive aliasing information through types, enabling strong updates to be performed soundly. Tanaka et al. [45] extended ConSORT to support arrays and pointer arithmetic. Our type system is a further extension of Tanaka et al.'s [45] to support nested arrays, in which the ownership of a pointer to an inner array can depend on the index of the outer array. We also implemented a prototype type inference tool that supports nested arrays, whereas the implementation by Tanaka et al. [45] only supports one-dimensional arrays.

Refinement type systems, such as liquid types [39, 38] (see [20] for a recent tutorial), express precise properties of a value using types. They have been applied to higher-order non-deterministic programs [49], class-based (functional) object-oriented languages [44], and smart contracts [31, 32, 33]. The systems mentioned above primarily target functional or smart-contract languages, and do not handle the imperative features that our type system does. Our type system also applies refinement types but handles these features through fractional ownership expressions.

The use of rational numbers to represent pointer usage originates with Boyland [5] and has been incorporated into permission accounting in separation logic [4], with extensions to concurrent settings [6]. Related type-theoretic treatments of resource usage appear in capability calculi [10, 9], though these use non-fractional capabilities. The fractional-permission approach has been applied to memory deallocation [43, 40], race freedom [46], concurrent resource deallocation [42], authenticity [24, 25, 11], and functional-correctness [30, 48] verification,[3] with a fraction attached to each reference variable or region. An earlier

[3] Although the type systems in these papers use fractions for counting *effects*, these effects can be viewed as permission to raise an event.

ownership-style analysis using integer-valued permissions in $\{0, 1\}$ was proposed by Heine and Lam [17] for memory leak detection. ConSORT and its extensions, including the present work, belong to the fractional-permission vein, attaching fractions to refinement-typed pointers.

Separation logic [37, 34, 18] underlies a range of verification tools that target rich functional-correctness specifications, including Smallfoot [2], VeriFast [19], bi-abduction-based shape analyses [7], Viper [29], Gobra [50], and CN [36]. The Iris framework [22] provides a higher-order concurrent separation logic in Coq on top of which such verification tools and meta-theoretical results can be built. These tools abstract the state of the heap using an assertion language, whereas ConSORT and its extensions, including ours, abstract how a pointer may be used using ownership types.

Deductive verifiers for Rust – Prusti [1], Creusot [14], and Verus [27] – together with RustBelt [21], which provides a mechanized semantic foundation for Rust's type system, rely on the ownership-and-borrow discipline that Rust enforces at compile time through its type system. Our target is a C-like language with raw pointer arithmetic where no such discipline is imposed, so aliasing must be controlled using fractional ownership.

DML [52, 51] uses types indexed by array length and position to check index-sensitive properties such as bounds safety, and does not reason about pointer aliasing. Tanaka et al. and ours reuse the idea of index-dependent refinement from this line, but imperative features are handled using fractional ownership.

8 Conclusion

In this paper, we proposed an extension to a type system featuring ownership and refinement types to support index-sensitive nested arrays, and we formally proved its soundness. We also implemented a verifier based on this type system and confirmed, through comparison with Tanaka et al.'s ConSORT implementation, that our verifier's capabilities are comparable. Furthermore, we successfully verified non-trivial properties of several programs involving nested arrays, a task not achievable with the previous work.

Our verifier requires type annotations on function arguments beyond simple types, even though ownership/predicate inference is theoretically possible without annotations. Improving the type-inference procedure in this regard is left for future work.

Another practical concern is the need for **alias** expressions. Our prototype already automates the insertion of **alias** expressions at syntactically determined points, and this mechanism eliminates the need for manual insertion of **alias** expressions in the majority of our benchmarks (Table 1). Nevertheless, certain patterns, particularly those involving non-trivial aliasing across function boundaries or multiple simultaneous redistributions, still require manual insertion. Extending the automated insertion to cover a wider range of aliasing patterns, as well as exploring more flexible ownership splitting strategies that would reduce the number of **alias** expressions needed in the first place, remain important directions for future work. We are looking at the direction of applying recent progress on demand-driven must-alias analysis, such as Boomerang [41] and its sparsification-based accelerations [23], to discharge **alias** annotations statically.

References

1 Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360573.

2 Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005. doi:10.1007/11804192_6.

3 Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015. doi:10.1007/978-3-319-23534-9_2.

4 Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 259–270. ACM, 2005. doi:10.1145/1040305.1040327.

5 John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003. doi:10.1007/3-540-44898-5_4.

6 James Brotherston, Diana Costa, Aquinas Hobor, and John Wickerson. Reasoning over permissions regions in concurrent separation logic. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 203–224. Springer, 2020. doi:10.1007/978-3-030-53291-8_13.

7 Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011. doi:10.1145/2049697.2049700.

8 Adrien Champion, Naoki Kobayashi, and Ryosuke Sato. HoIce: An ICE-based non-linear Horn clause solver. In Sukyoung Ryu, editor, *Programming Languages and Systems*, pages 146–156, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-02768-1_8.

9 Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 213–224, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1411204.1411235.

10 Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 262–275, New York, NY, USA, 1999. Association for Computing Machinery. doi:10.1145/292540.292564.

11 Morten Dahl, Naoki Kobayashi, Yunde Sun, and Hans Hüttel. Type-based automated verification of authenticity in asymmetric cryptographic protocols. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, volume 6996 of *Lecture Notes in Computer Science*, pages 75–89. Springer, 2011. doi:10.1007/978-3-642-24372-1_7.

12 Emanuele De Angelis, Fabio Fioravanti, John Patrick Gallagher, Manuel V. Hermenegildo, Alberto Pettorossi, and Maurizio Proietti. Analysis and transformation of constrained Horn clauses for program verification. *Theory and Practice of Logic Programming*, 22(6):974–1042, 2022. doi:10.1017/S1471068421000211.

13 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1145/357119.

14 Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for the deductive verification of Rust programs. In *Formal Methods and Software Engineering: 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24–27, 2022, Proceedings*, pages 90–105, Berlin, Heidelberg, 2022. Springer-Verlag. doi:10.1007/978-3-031-17244-1_6.

15 Yusuke Fujiwara, Yusuke Matsushita, Kohei Suenaga, and Atsushi Igarashi. Ownership refinement types for pointer arithmetic and nested arrays, 2026. arXiv:2604.22361.

16 Kodai Hashimoto and Hiroshi Unno. Refinement type inference via Horn constraint optimization. In *Static Analysis - 22nd International Symposium, SAS 2015, Venice, Italy, September 9-11, 2015, Proceedings*, volume 9215 of *Lecture Notes in Computer Science*, pages 199–216. Springer, 2015. doi:10.1007/978-3-662-48288-9_12.

17 David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In Ron Cytron and Rajiv Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 168–181. ACM, 2003. doi:10.1145/781131.781150.

18 Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 14–26. ACM, 2001. doi:10.1145/360204.375719.

19 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011. doi:10.1007/978-3-642-20398-5_4.

20 Ranjit Jhala and Niki Vazou. Refinement types: A tutorial. *Found. Trends Program. Lang.*, 6(3–4):159–317, 2021. doi:10.1561/2500000032.

21 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, 2018. doi:10.1145/3158154.

22 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015. doi:10.1145/2676726.2676980.

23 Kadiray Karakaya and Eric Bodden. Two sparsification strategies for accelerating demand-driven pointer analysis. In *IEEE Conference on Software Testing, Verification and Validation (ICST 2023)*, pages 305–316. IEEE, 2023. doi:10.1109/ICST57152.2023.00037.

24 Daisuke Kikuchi and Naoki Kobayashi. Type-based verification of correspondence assertions for communication protocols. In Zhong Shao, editor, *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*, volume 4807 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2007. doi:10.1007/978-3-540-76637-7_13.

25 Daisuke Kikuchi and Naoki Kobayashi. Type-based automated verification of authenticity in cryptographic protocols. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2009. doi:10.1007/978-3-642-00590-9_17.

26 Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 2010. doi:10.1145/1667048.1667051.

27 Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023. doi:10.1145/3586037.

28 Ryoya Mukai, Naoki Kobayashi, and Ryosuke Sato. Parameterized recursive refinement types for automated program verification. In *Static Analysis: 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5–7, 2022, Proceedings*, volume 13790 of *Lecture Notes in Computer Science*, pages 397–421. Springer, 2022. doi:10.1007/978-3-031-22308-2_18.

29 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, Lecture Notes in Computer Science, pages 41–62. Springer, 2016. doi:10.1007/978-3-662-49122-5_2.

30 Takashi Nakayama, Yusuke Matsushita, Ken Sakayori, Ryosuke Sato, and Naoki Kobayashi. Borrowable fractional ownership types for verification. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II*, volume 14500 of *Lecture Notes in Computer Science*, pages 224–246. Springer, 2024. doi:10.1007/978-3-031-50521-8_11.

31 Yuki Nishida, Hiromasa Saito, Ran Chen, Akira Kawata, Jun Furuse, Kohei Suenaga, and Atsushi Igarashi. Helmholtz: A verifier for Tezos smart contracts based on refinement types. In *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part II*, pages 262–280, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-72013-1_14.

32 Yuki Nishida, Hiromasa Saito, Ran Chen, Akira Kawata, Jun Furuse, Kohei Suenaga, and Atsushi Igarashi. Helmholtz: A verifier for tezos smart contracts based on refinement types. *New Gener. Comput.*, 40(2):507–540, 2022. doi:10.1007/S00354-022-00167-1.

33 Yuki Nishida, Kohei Suenaga, and Atsushi Igarashi. iCon: Automated verification of inter-transaction properties in Tezos smart contracts with unknowns. In *Proceedings of 2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC 2024)*, 2024.

34 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. doi:10.1007/3-540-44802-0_1.

35 Mário Pereira and António Ravara. Cameleer: A deductive verification tool for OCaml. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 677–689. Springer, 2021. doi:10.1007/978-3-030-81688-9_31.

36 Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. CN: Verifying systems C code with separation-logic refinement types. *PACMPL*, 7(POPL), 2023. doi:10.1145/3571194.

37 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.

38 Patrick Maxim Rondon. *Liquid Types*. PhD thesis, University of California, San Diego, 2012. Merritt ID: ark:/20775/bb72709827. URL: https://escholarship.org/uc/item/1646v8mx.

39 Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169. ACM, 2008. doi:10.1145/1375581.1375602.

40 Tatsuya Sonobe, Kohei Suenaga, and Atsushi Igarashi. Automatic memory management based on program transformation using ownership. In Jacques Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *Lecture Notes in Computer Science*, pages 58–77. Springer, 2014. doi:10.1007/978-3-319-12736-1_4.

41 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *LIPIcs*, pages 22:1–22:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.22.

42 Kohei Suenaga, Ryota Fukuda, and Atsushi Igarashi. Type-based safe resource deallocation for shared-memory concurrency. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 1–20. ACM, 2012. doi:10.1145/2384616.2384618.

43 Kohei Suenaga and Naoki Kobayashi. Fractional ownerships for safe memory deallocation. In Zhenjiang Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 128–143. Springer, 2009. doi:10.1007/978-3-642-10672-9_11.

44 Ke Sun, Di Wang, Sheng Chen, Meng Wang, and Dan Hao. Formalizing, Mechanizing, and Verifying Class-Based Refinement Types. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 39:1–39:30, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2024.39.

45 Izumi Tanaka, Ken Sakayori, and Naoki Kobayashi. Ownership types for verification of programs with pointer arithmetic. In Gabriele Keller and Meng Wang, editors, *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, PEPM 2024, London, UK, 16 January 2024*, pages 94–106. ACM, 2024. doi:10.1145/3635800.3636965.

46 Tachio Terauchi. Checking race freedom via linear programming. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 1–10. ACM, 2008. doi:10.1145/1375581.1375583.

47 John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. ConSORT: Context- and flow-sensitive ownership refinement types for imperative programs. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 684–714. Springer, 2020. doi:10.1007/978-3-030-44914-8_25.

48 Hideto Ueno, John Toman, Naoki Kobayashi, and Takeshi Tsukada. Counterexample generation for program verification based on ownership refinement types. In Sam Lindley and Torben Æ. Mogensen, editors, *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2021, Virtual Event, Denmark, January 18-19, 2021*, pages 44–57. ACM, 2021. doi:10.1145/3441296.3441396.

49 Hiroshi Unno, Yuki Satake, and Tachio Terauchi. Relatively complete refinement type system for verification of higher-order non-deterministic programs. *Proc. ACM Program. Lang.*, 2(POPL), 2018. doi:10.1145/3158100.

50 Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. Gobra: Modular specification and verification of Go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, Lecture Notes in Computer Science, pages 367–379. Springer, 2021. doi:10.1007/978-3-030-81685-8_17.

51 Hongwei Xi. Dependent ML: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007. doi:10.1017/S0956796806006216.

52 Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 214–227. ACM, 1999. doi:10.1145/292540.292560.