CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Minimal Language for Mutable Data Structures

Master's thesis in Computer Science and Engineering

Sarah McShane

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

A Minimal Language for Mutable Data Structures

Sarah McShane

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

A Minimal Language for Mutable Data Structures
Sarah McShane

Supervisor: Jonas Almström Duregård, Computer Science and Engineering
Examiner: Ana Bove, Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

A Minimal Language for Mutable Data Structures
Sarah McShane
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis presents the design of a minimal imperative language for reasoning about mutable data structures and memory effects. The language separates variable introduction, alias creation, and mutation into distinct operators and makes allocation behavior explicit in the program text. Since locations are introduced only through visible constructs, sharing and memory growth can be reasoned about from the source code. The results show how different construction patterns produce distinct memory topologies, traversal styles, update behavior and sharing relationships. The thesis also explores the limitations of a simple structural type system in combination with mutable linked structures.

Keywords: Programming languages, mutable data structures, language design, memory semantics, operational semantics, structural typing

Glossary

Aliasing	Multiple names referring to the same memory location.
Allocation	Creation of a fresh location in the store.
Cell-level mutation	Mutation that updates a primitive value without changing any surrounding structures.
Compatibility relation	The full compatibility predicate Compatible(τ_1, τ_2) including structural equality and the relaxed cases. Used by the write operator to determine whether a value of one type may replace a value of another.
Compound value	A value composed of multiple components (containing either locations or values).
Deep copy	Copying the outer location together with the values of the inner locations.
Dereference	Operation that follows a pointer to obtain the referenced location.
Embedded construction	Construction pattern where compound values are embedded within larger values.
Environment	Mapping from variables to locations.
Finite unfolding	A representation of a linked structure obtained by expanding it to a finite depth.
Graph-shaped linked list	A linked-list representation whose nodes are individually allocated and connected through pointers.
Hybrid linked list	A linked-list representation containing both copied structure and shared locations.

In-place mutation	Updating an existing structure without allocating a replacement.
Inner location	A location referenced from within a compound value.
Location	Mutable cell in the store.
Lvalue	An expression that denotes a location in the store.
Memory topology	The connections between locations in the store.
Mutation	Updating the value stored at an existing location.
Operational structure	The kind of value stored at a location.
Outer location	The top-level location containing a compound value.
Outer structure	The compound value stored at an outer location.
Precision loss	A situation in which the type checker's information no longer accurately describes the type of the structure reached at runtime.
Primitive value	A non-compound value such as an integer, boolean, address or null value.
Referential construction	Construction pattern where compound values are connected through explicit pointers.
Relaxation	Extension of the compatibility relation beyond strict structural equality for validated linked-list types. It consists of a shallow compatibility rule and a null-to-pointer compatibility rule.
Shallow compatibility rule	A rule within the compatibility relation that treats two pointers to validated linked-list types as compatible regardless of their unfolding depth.
Shallow copy	Copying only the outer location, resulting in sharing of existing inner locations.
Sharing	Multiple structures referring to the same location. In contrast to aliasing, sharing is not visible in the environment.
State	The current contents of the environment and store.

Store	Mapping from locations to values.
Structural type system	A type system where compatibility is determined by type structure rather than type names.
Tree-shaped linked list	A linked-list representation where each node contains the next node.

Contents

1

Introduction

Research in assignment semantics and computer science education has found that implicit semantics and overloaded operators make reasoning about effects such as aliasing and mutation difficult [1, 2, 3]. While high-level languages provide convenient abstractions that reduce implementation complexity, the mechanisms underlying copying, aliasing and mutation are often hidden behind a small number of overloaded operators, creating a gap between the surface structure of programs and the memory effects they produce.

For example, in Python an assignment can result in either shared references or independent values depending on the objects involved and the context in which the assignment occurs. Similar behaviors appear across many high-level programming languages, where memory effects depend on language-specific rules and implicit semantics. As a consequence, reasoning about program behavior often requires understanding how a language interprets and manages relationships between variables, values, and memory.

Hidden semantics present a particular challenge for novice programmers. Studies in computer science education have found that students using high-level languages frequently struggle to determine when variables share state, whether a value has been copied or aliased, and when an update performed through one variable affects another part of the program. These difficulties become especially relevant in introductory data structures and algorithms courses, where understanding mutable data structures requires reasoning about dynamic program state and memory effects. Researchers have observed that students often form incomplete or incorrect mental models of assignment and memory behavior, leading to confusion when program execution does not match expectations [3].

How easily programmers can reason about the behavior of programs depends partly on the design choices and which abstractions languages provide. Hoare argues that language design should prioritize simplicity to a degree that allows programmers to understand the effects and consequences of individual language constructs, so that intellectual effort can be fully directed to solving the problem [4]. As an illustration, he points to small assembly languages, where each instruction has a well-defined and locally understandable effect that can be reasoned about independently of other

instructions. Although Hoare's discussion is mostly concerned with the design of practical programming languages, his argument that programmers should be able to understand the effect of every decision they make is directly relevant to the problem this thesis tries to address. If understanding program behavior depends on remembering a large number of special cases, hidden interactions, and context-dependent rules, reasoning about programs becomes more difficult. The present work therefore explores whether some of the mechanisms that make mutable data structures difficult to reason about can be exposed through syntax.

Taking inspiration from minimal calculi and other small languages intended for specific domains, this thesis presents a minimal imperative language with three core operators whose effects on the store are distinct and non-overlapping. The operators are **assign**, which introduces a new variable by allocating a fresh location and binding the variable to it; **alias**, which creates a new name for an existing location; and **write**, which updates the contents of an existing location. The language also makes allocation behavior of expressions visible in programs.

The thesis addresses the following research questions:

- **RQ1:** How can separating allocation, aliasing, and mutation into distinct syntactic operators support reasoning about mutable data structures and memory effects in a minimal imperative language?
- **RQ2:** How do different syntactic construction patterns induce different memory topologies and sharing behavior?
- **RQ3:** What limitations emerge when using a simple structural type system together with mutable pointer-based structures?

1.1 Limitations

The language presented in this thesis is intentionally narrow in scope, focusing only on the core mechanisms underlying basic mutable data structures. Functions, scope and recursion are not supported, which limits the range of data structures and algorithms that can be expressed.

The type system is intentionally simple and cannot always preserve precise type information during mutation of mutable linked structures. Certain pointer-based structures such as doubly linked lists are therefore not supported, since they would introduce further special cases in order to be expressed safely. No soundness proof has been produced for the type system.

The evaluation is based on executable examples and semantic analysis rather than formal verification. The design explored in this thesis should be understood as one possible approach to making implicit semantics and hidden memory effects more explicit.

1.2 Thesis Overview

Chapter 2 presents background on reasoning about mutable data structures and memory effects, implicit semantics, levels of abstraction, and related work on minimal and specialized programming languages. Chapter 3 describes the research methodology and the iterative language development process. Chapter 4 presents the final language design and introduces its core constructs. Chapter 5 formalizes the language, presenting its syntax, big-step operational semantics and typing rules. Chapter 6 presents the runtime model through concrete program examples that illustrate the separation of memory effects. Chapter 7 presents the results, demonstrating how different syntactic construction patterns induce different memory topology, sharing behavior, traversal strategies, and mutation effects. The chapter also demonstrates the language's expressiveness through a representative set of mutable data-structure operations and algorithms. Chapter 8 discusses the results in relation to the research questions and previous literature. Finally, Chapter 9 concludes the thesis and outlines directions for future work.

2

Background

This chapter provides background for the language design and methodological choices presented later in the thesis. Sections 2.1 and 2.2 review previous work on difficulties in reasoning about mutable data structures and the role of abstraction, providing an informal domain analysis that defines the scope of the project. Section 2.3 discusses language design choices related to hidden semantics, assignment behavior and explicit memory effects, which motivate the core design principles and goals of the language. Section 2.4 reviews work on small, specialized programming languages and minimal calculi, placing the methodological approach adopted in this thesis in the context of existing work.

2.1 Difficulties in reasoning about data structures

Research on difficulties students encounter in their first data structures course is relatively limited [5, 6]. However, existing studies consistently identify reasoning about state, references and structural behavior as significant challenges. Mazumder et al. identify reasoning about state changes as a threshold concept in computer science and argue that arrays contain enough dynamic behavior to require understanding of several complex semantic concepts [7]. They note that arrays are foundational for understanding implementations of standard data structures and algorithms such as sorting algorithms, binary search and hashing functions. Although arrays are often introduced as simple indexed collections, reasoning about them still requires understanding assignments, memory layout, indexing and how program state changes over time. Without understanding that arrays represent mutable regions of memory, reasoning about resizing costs, element shifting and in-place updates becomes difficult.

Similar findings are reported by Zingaro et al. who investigated students' difficulties with arrays, linked lists and binary search trees [6]. The authors identified mistakes such as unnecessarily traversing a linked list to locate the tail when a pointer to it was available, failing to attach newly allocated nodes correctly and ignoring performance implications of operations. Related difficulties were also found for generalized arrays and binary search trees. The authors speculate that these errors may indicate that students often memorize operational procedures for manipulating data structures

without fully understanding the underlying structural relationships and invariants [6].

A study conducted by Layman et al. examines difficulties encountered in Python-based data structures courses where students were not allowed to use Python's built-in structures and instead had to implement the data structures themselves. The authors found that linked lists were considered among the most difficult structures to implement, particularly because students struggled with variables as references and with explicit reference manipulation [5]. The study also reports substantial differences between students' conceptual understanding of data structures and their ability to reason about implementation-level behavior.

Research on Python as an introductory programming language more broadly identifies references and shared mutable state as major sources of confusion. Johnson et al. report that many students struggle to understand that modifying one variable may also affect another variable referring to the same underlying structure [3]. Their findings support earlier work identifying references as a threshold concept in computer science, since reasoning about mutation, copying and aliasing requires understanding that variables may share memory locations rather than store independent values.

Johnson et al. further argue that extensive operator overloading hides important semantic distinctions related to memory behavior and causes significant misunderstandings when reasoning about mutable data structures [3]. Operations that appear syntactically similar may produce fundamentally different effects on memory and sharing behavior. For example, expressions such as `a = a + b` and `a += b` may produce the same final value while differing operationally in whether a new structure is allocated or an existing structure is modified in place. Understanding these behaviors requires reasoning about memory locations, references, sharing, copying and mutability, even though these effects are not visible in the syntax itself. The study suggests that operator overloading hides important semantic differences behind uniform notation, obscuring the role of references, copying and mutation.

Notional machines are abstract models of program execution intended to help programmers understand how program state evolves during execution. Fincher et al. argue that notional machines play an important role in computing education by directing attention toward aspects of program behavior that are not immediately visible in the program text [8]. If these underlying mechanisms remain hidden, learners may construct incorrect mental models of how programs execute. Understanding mutable data structures therefore depends not only on the behavior of the language itself, but also on whether the notional machine suggested by the syntax accurately reflects the underlying execution model.

When operators are overloaded with fundamentally different semantic meanings, the notional machine implied by the syntax diverges from the actual execution, which may cause students to form an incorrect mental model from the syntax. For example, writing `a = b` in Python suggests that `a` gets the value of `b`, while the actual execution

model says that both names refer to the same object. The syntax draws attention to the wrong thing, directing the learner toward a value-copying model when the actual behavior is reference sharing. Johnson et al. argue that Python's simple syntax hides a complex notional machine and that suitable programming languages for computer science education should support the construction of accurate mental models directly from syntax and execution behavior [3]. Mazumder et al. similarly argue that notional machines can help learners understand dynamic state-changing aspects of programming by making the underlying operational model explicit [7].

Research on student difficulties in upper-level computer science courses identifies scope, mutation and aliasing as persistent sources of difficulty even after significant programming experience, suggesting that the problem is not confined to novice programmers. Fisler et al. studying students' understanding of Java and Scheme semantics argue that extended exposure to a programming language is insufficient for developing accurate semantic models of program behavior, particularly in the presence of mutation [9]. Together, the studies in this section suggest that difficulties with mutable data structures persist across experience levels and are not eliminated by increased programming practice alone. They point toward the same underlying problem of programmers struggling to form accurate models of how programs interact with memory when the memory effects and semantics of operations are hidden.

2.2 Abstraction

Many difficulties associated with basic mutable data structures concern reasoning across different levels of abstraction. Learners must simultaneously reason about high-level structural properties and low-level operational behavior, including allocation, references, mutation and changing memory relationships. This is further supported by Sibia et al. who describe the problem as one of structural abstraction, where programmers may understand individual execution steps but struggle to connect low-level memory operations to higher-level structural changes in mutable data structures [10]. In particular, reasoning about lists, trees and recursive structures requires coordinating multiple levels of abstraction simultaneously, including state changes, memory relationships and structural behavior. This is especially relevant to mutable data structures, where local updates may propagate through shared structure in non-obvious ways.

The connection between structure and abstraction is also discussed by Nakar et al. who similarly argue that understanding data structures requires moving between different levels of abstraction, from high-level views of operations and organization to low-level implementation details [11]. The authors note that students often struggle to coordinate these different perspectives, making abstraction itself an educational challenge rather than merely a tool for managing complexity.

A related perspective is provided by Jarc, who proposes a framework organizing data structures into four levels of increasing abstraction [12]. The first level consists only of collections of bits stored at addresses. The second level distinguishes between

statically allocated and dynamically allocated structures, where arrays and records form the basis for sequential structures, while pointers allow construction of linked structures. The third level contains structures such as lists, trees and graphs, while the fourth level contains more abstract structures such as stacks, priority queues and keyed tables. Jarc argues that each level introduces qualitatively different reasoning requirements and that properties essential at one level differ from those at another [12]. He further connects these abstraction levels to different programming language traditions, where low-level imperative languages correspond more closely to arrays, records and pointers, while higher-level languages increasingly abstract away memory representation and allocation behavior.

This framework helps contextualize many of the previously discussed student difficulties. Problems involving references, mutation, sharing and linked structures arise partly because students must reason simultaneously about structural abstractions and the lower-level memory operations implementing them. In many standard high-level languages, allocation behavior, sharing relationships and memory topology are largely implicit, being hidden behind object models and runtime systems. While this supports concise programming, it may also obscure the operational mechanisms underlying mutable data structures. These observations motivate examining mutable data structures at a level where allocation behavior, references and structural relationships remain operationally visible.

2.3 Hidden semantics

Many programming languages support forms of reference semantics, where multiple names may refer to the same underlying memory location rather than storing independent copies of a value. Under such semantics, updating one reference may indirectly affect other parts of a program through shared mutable state. Reasoning about mutation therefore requires understanding aliasing relationships and memory sharing behavior. Batdalov and Nikiforova describe this phenomenon using the term referential assignment, which they define as assignment behavior where multiple symbols may refer to the same underlying location [13]. They contrast this with value assignment, where assignment creates an independent copy of a value, and partial assignment, where only part of a compound value is updated. These different assignment semantics expose different tradeoffs between copying cost, sharing behavior and mutation visibility.

A particularly relevant case is shallow copying, which combines behaviors of value assignment and reference assignment [13]. A shallow copy allocates a new outer structure, which gives the appearance of an independent value, but the inner locations of the copy are shared with the original. The result is that the copy is independent at the surface level but shared at the level of its internal structure. Mutations to the inner components of the original therefore propagate to the copy, and vice versa, without any visible indication that this will happen.

Research on mutable value semantics has identified implicit aliasing (called *sharing*

in this thesis) as a significant obstacle to local reasoning about program behavior [2]. Local reasoning is the ability to understand a program by considering only the memory locations accessed by an operation while assuming that unrelated locations remain unchanged [14]. In other words, reasoning locally means being able to understand what a small fragment of code will change without having to reason about the entire program state. Sharing makes this difficult because updates may affect memory through references that are not visible at the point where the update occurs. Racordon et al. describe this phenomenon as "spooky action at a distance", arguing that hidden sharing relationships make both human reasoning and program analysis significantly more difficult [2]. This problem is particularly relevant for imperative programs manipulating mutable data structures, where understanding the effects of mutation depends on reasoning about sharing relationships and store structure.

Language design choices influence how programmers reason about memory, mutation and sharing. Assignment semantics have been identified as a particular source of ambiguity in imperative programming languages [15, 1, 2]. While the previous section discussed overloaded operators in general, assignment is a particularly important case since it is the primary mechanism through which variables acquire values and relationships to memory, and it appears in nearly every program. When a single assignment operator performs different operations depending on the runtime representation, the programmer cannot determine whether an operation creates a new value, introduces sharing, or modifies an existing location.

Racordon and Buchs argue that the ambiguity of assignment semantics does not stem from supporting multiple assignment strategies themselves, but from hiding semantically different behaviors behind implicit or overloaded assignment operations [1]. They further note that aliasing appears consistently as a source of difficulty, and that overloading assignment operators steepens the learning curve for beginners [1]. To address these issues they designed Anzen, a multi-paradigm general-purpose language that separates aliasing, deep copying and moving into distinct assignment operators to simplify reasoning about assignments [1]. Its aliasing operator creates a shared reference to an existing object, its copy operator performs a deep copy and mutates the target in place, and its move operator transfers ownership without duplication. The authors argue that these explicit operators return control over memory behavior to the programmer by making copying and aliasing intentional choices rather than implicit effects.

The idea that semantically distinct language constructs should be represented explicitly in the syntax also appears in Grace, an educational language for teaching object-oriented programming [15]. Grace separates concepts such as variable declaration, constant declaration, reassignment and equality into distinct operators, which the authors argue improve semantic clarity, even at the cost of slightly increased syntactic overhead.

The language presented in this thesis operates in the same design space as Anzen

and Grace but has the goal of making the memory effects underlying basic mutable data structures visible. Like these languages, it separates operations that are often combined under a single assignment construct. But rather than separating deep copying and ownership transfer as in Anzen, or distinguishing declarations and reassignment as in Grace, the present design provides an operator that allocates a fresh location and binds a variable, an operator that creates an alias to an existing location, and an operator that overwrites an existing location.

2.4 Specialized languages and minimal calculi

Small, specialized programming languages appear across computer science in different forms depending on their context and purpose. In software engineering, such languages are commonly referred to as domain-specific languages. In computer science education research, languages designed for teaching specific concepts are often described as educational or teaching languages. In programming language research and formal methods, small language artifacts designed for studying properties or isolating essential features are commonly referred to as core languages, minimal cores or calculi. While terminology and intended use differ between domains, these approaches share several important design principles: restricting the language to a narrow problem domain, reducing unnecessary complexity and exposing the concepts considered most relevant to the problem being studied.

One characteristic that often distinguishes domain-specific languages from general-purpose languages is that the latter typically support sophisticated user-defined abstractions and broad computational expressiveness, while domain-specific languages intentionally restrict these features. The purpose of a domain-specific language is instead to reduce the semantic distance between a problem domain and its formal representation [16]. Rather than attempting to support all forms of programming equally well, domain-specific languages are optimized around a smaller set of concepts and operations relevant to a specific domain. What counts as relevant varies considerably between domains, but often concerns whether important concepts can be expressed directly and concisely through the core abstractions of the language itself.

Empirical evidence suggests that restricting a language to a narrow domain may improve program comprehension. Kosar et al. found that participants achieved higher correctness on program understanding tasks when using domain-specific languages compared to general-purpose languages across several domains, including graph description, feature modeling and graphical user interfaces [17]. This suggests that reducing irrelevant general-purpose complexity may support reasoning within a focused problem domain. Similar ideas appear in educational language design, such as Grace, which emphasizes conceptual clarity over industrial-scale language features [15]. The designers of Grace argue that novice programmers benefit from focusing on fundamental concepts before encountering the additional complexity of large-scale software engineering concerns [15].

Another form of small, specialized languages is minimal core languages, or calculi, used in programming language research. While notional machines remove irrelevant implementation machinery in order to explain execution, minimal cores remove irrelevant semantic machinery in order to study particular language mechanisms formally. Both traditions share the underlying principle that exposing only the essential mechanisms of a domain simplifies reasoning. Rather than serving as programming languages for practical use, minimal cores are designed to isolate specific semantic mechanisms while abstracting away features considered irrelevant to the property being studied.

A well-known example of a minimal core is Featherweight Java, which was designed to isolate the essential mechanisms of Java's type system [18]. Featherweight Java preserves the central semantic structure of Java while omitting nearly all practical language features, containing only classes, methods, fields, inheritance and dynamic typecasts. The authors argue that reducing Java to a small formal core makes rigorous reasoning about properties such as type safety significantly easier [18].

Several minimal imperative calculi have also been developed. Pierce's Imp is used to teach formal reasoning and program verification through proof assistants [19], while Winskel's IMP introduces operational semantics using states and transition systems [20]. IMP programs are while-programs consisting of arithmetic expressions, boolean expressions and commands that transform program state through execution. The execution of a program is modeled as a sequence of state transformations. This style of explicit state-based modeling is particularly relevant for reasoning about mutable data structures, since data structure behavior depends fundamentally on how memory relationships change as execution proceeds.

As the examples above illustrate, constructing small formal languages to isolate particular semantic phenomena is a standard methodology in programming language research. Such minimal calculi and core languages allow precise reasoning about selected mechanisms by abstracting away unrelated language features and implementation complexity. Instead of attempting to replace existing general-purpose languages used in data structures education, this project designs a minimal language containing only the semantic primitives relevant to reasoning about mutable data structures and memory effects.

3

Methodology

3.1 Research approach

This project follows an exploratory semantics-oriented language design approach based on the iterative construction of a minimal imperative language for reasoning about memory behavior in basic data structures. The purpose has been to identify which semantic mechanisms make reasoning about mutable data structures difficult in high-level languages, and how these mechanisms can be isolated and exposed explicitly through a small formal language.

The project began with a literature review to identify difficulties encountered in introductory data structures and algorithms courses, particularly in courses taught using high-level languages. The background chapter discusses these findings in more detail. The initial review of data structures education literature identified recurring difficulties involving aliasing, references, mutation, hidden semantics, overloaded operators and state changes. Because these concepts also appear outside the context of data structures courses, the search was expanded to include related areas, such as novice programmers' difficulties in high-level languages. Many of these studies reported similar difficulties, suggesting that the challenges observed in data structures education are part of a broader problem of reasoning about mutable program state.

Several approaches were considered during the early stages, including domain-specific languages, educational teaching languages, visualization tools and minimal calculi. Since the difficulties identified are not confined to any single language or course, but reflect a general pattern in how high-level languages abstract over memory behavior, the approach adopted here is inspired by minimal calculi in programming language research, where unnecessary language features are removed in order to isolate the core semantic mechanisms relevant to the target domain [18, 20]. However, instead of using a formal calculus notation, the language is expressed in a concrete programming-language syntax that exposes memory effects directly in the program text. The review of small languages, domain-specific languages and minimal calculi discussed in the background chapter informed these methodological choices.

The project combines operational semantics and iterative language design. The

semantics were developed incrementally through multiple prototypes implemented in Haskell, which served as executable semantic models for exploring how different design choices affected sharing behavior, mutation and observable store structure, with the goal of making the language's semantics explicit through the syntax. The interpreter adopts IMP's big-step evaluation style. Winskel's IMP served as an initial reference point for the language due to its small imperative core [20], but the project diverged early from standard IMP-style semantics since expressing aliasing requires separation between environment and store.

The resulting language should be understood as a minimal semantic model for reasoning about memory effects and behavior in the context of mutable data structures, rather than as a practical programming language or teaching tool.

3.2 Domain analysis

The domain was identified through an informal analysis of previous research concerning conceptual difficulties in data structures education and related areas of programming. This analysis was used to identify constraints that guided the subsequent design of the language. The identification focused on determining which data structures and semantic concepts a minimal reasoning core should be capable of expressing, and what underlying semantic phenomena make reasoning about these concepts difficult. As discussed in the background chapter implicit semantics, overloaded operators and assignment semantics were identified as recurrent sources of difficulty, which motivated the decision to make distinct semantic behaviors syntactically explicit.

The domain analysis focused on identifying:

- which basic mutable data structures should be representable,
- which memory behaviors were essential to model explicitly,
- and which operations were necessary to express those behaviors.

The scope was restricted to the semantic mechanisms necessary for reasoning about basic data structures such as arrays, records and linked lists. This choice was influenced by Jarc's framework of data structure abstraction levels, where arrays and records together with pointers form the basis for more complex dynamic structures such as linked lists, trees and graphs [12]. Since the goal was to reason about aliasing, sharing, mutation and memory structure, the language focuses on the level where these relationships become explicit while avoiding additional complexity not relevant to the problem domain. This level of abstraction is also sufficient to support standard algorithms such as insertion sort, bubble sort, and linked-list operations, all of which rely on mechanisms identified in the background, including in-place mutation, traversal of linked structures, and explicit reference manipulation.

These requirements determine the minimal set of primitive operations necessary for representing the relevant memory behaviors. The language is intentionally built around a very small number of semantically distinct operations together with a minimal set of constructs for representing memory structure. Removing any of the core operators would eliminate the ability to represent key aspects of the target domain. Removing the assign operator would make it impossible to introduce fresh variable bindings, since it is the only operator that allocates a fresh location and binds a variable to it. Removing the alias operator would remove the ability to introduce explicit aliases between existing locations, which is one of the core phenomena the language is designed to make visible. Removing the write operator would make the language immutable. Removing the language constructs for pointers and dereference would prevent the representation of pointer-based linked lists. The remaining language constructs largely form syntactic or arithmetic categories included for convenience rather than additional semantic expressiveness.

3.3 Iterative semantic design

The language was developed through an iterative refinement process in which multiple semantic prototypes were implemented and compared using a shared test suite. Several alternative semantics were explored while preserving a shared core design which separates the introduction of variables, creation of aliases and writing to existing locations into three distinct operators. All three prototypes also share the same environment-store model. The prototypes mainly differ in how compound values such as arrays and records handle internal allocation and sharing.

3.3.1 Values prototype

The values prototype modeled arrays and records as containing values rather than locations. This resulted in a value-oriented semantics where copying duplicated entire structures. The prototype was comparatively easy to reason about since structures remained independent unless pointers were introduced manually. However, this prototype was less expressive than the other two, and could not represent sharing or aliasing within compound values. Furthermore, updating a component of a compound value reconstructed the enclosing value even though the syntax appeared to perform a direct update, which conflicted with the design principle that semantics should be exposed through syntax.

3.3.2 Reference prototype

The reference prototype explored a more reference-oriented semantics in which arrays and records contained locations rather than values. In this prototype, constructing a compound value from existing variables reused previously allocated locations instead of allocating fresh inner locations, introducing implicit context-dependent sharing. The same syntactic operation could either allocate fresh structure or introduce sharing depending on whether the right-hand side expression contained existing variables. This is the kind of implicit behavior identified in previous studies as a

source of confusion in standard languages, where the effect of an operation cannot be determined from the syntax alone. The reference prototype therefore conflicted with the design principle that effects should not be hidden and that each core operator should have a distinct, context-independent meaning.

3.3.3 Locations prototype

The locations prototype, which became the primary design, preserves expressiveness and visibility of sharing by avoiding implicit or conditional allocation. Arrays and records allocate fresh inner locations whenever they are constructed, independent of the values used during construction. Sharing therefore never arises implicitly. Instead, it occurs only when previously allocated locations are reused through explicit pointers or through copying structures that contain locations. Adding a syntactic marker to indicate that structures contain inner locations further makes sharing relationships inferable from the program text.

3.4 Prototype implementation

The prototypes were implemented as executable interpreters in Haskell using big-step evaluation. Big-step semantics was chosen because the primary goal was to reason about resulting memory states and observable store structure rather than intermediate reduction steps.

The concrete and abstract syntax were specified using BNFC, which generated the parser and abstract syntax tree. The interpreter and type checker were validated using an incremental test suite of small programs designed to test allocation, aliasing, mutation, pointer behavior and control-flow edge cases. Several issues related to linked lists, pointer dereference, control flow and making memory effects explicit were discovered and addressed during this process.

The interpreters model execution using an explicit environment-store representation: variables are mapped to locations through the environment, while locations are mapped to values through the store. This separation was necessary from the beginning of the design process since reasoning about aliasing, mutation and sharing required direct visibility into how structures were allocated and connected. Initially, the explicit store representation functioned primarily as a mechanism for observing otherwise hidden sharing behavior. Later refinements increasingly moved these effects into the syntax itself.

The executable prototypes were repeatedly used to test semantic invariants, analyze edge cases and refine both the operational semantics and typing rules.

3.5 Evaluation

The evaluation examines whether the resulting language satisfies the design goals identified during the domain analysis, and whether the design exposes the memory effects underlying mutable data structures. Evaluation of the final design was carried out through executable examples implemented using the Haskell interpreter for the locations prototype. The resulting environment-store states were then analyzed to evaluate the language's behavior.

The evaluation consists of three parts. The first part uses small semantic examples to isolate specific behaviors, including allocation, copying, sharing, aliasing and mutation, demonstrating how individual operators affect program state. The second part examines how different construction patterns give rise to different linked-list representations, showing how memory topology and traversal style emerge from syntactic choices. The third part examines the limitations of the structural type system through examples involving mutable pointer-based structures, identifying cases where the type system loses precision and where a more precise typing approach would be required. These examples also reflect a broader evaluation process in which executable examples were used throughout the design process to test semantic consistency, identify edge cases and incrementally refine both the language and the type checker.

In addition to these semantic examples, the language was evaluated using a small set of representative data structure operations and algorithms to demonstrate that it is expressive enough to represent standard mutable data-structure manipulations despite its minimal design.

4

Language Design

This chapter presents the final language design. The design targets the abstraction level identified by Jarc where arrays, records and pointers serve as the building blocks for dynamically allocated structures [12].

4.1 Design goals and design principles

The language was designed around a set of semantic principles intended to make memory effects explicit and minimize hidden semantics. These principles are:

- each core operator should have a single, context-independent meaning,
- different operators should produce observably distinct effects on program state,
- semantics should be made visible through syntax whenever possible,
- locations should preserve their operational structure as execution proceeds.

Locations preserving their operational structure refers to the kind of value stored at a location as execution progresses. While the type system does not assign types to locations, the idea is inspired by Pierce's notion of store typing [21]. Ideally, a location initially storing an integer should continue to store integer values, while a location initially storing a record should continue to store record values of the same size.

Many properties of the language follow directly from these principles. In particular, the separation between assign (`:=`), alias (`->`) and write (`=`) follows from the principles that different memory effects should be distinct and visible in syntax. This addresses the problem identified by Johnson et al. where a single overloaded operator can implicitly perform different operations depending on context [3]. The resulting language makes the mechanisms underlying mutable data structures visible while remaining intentionally minimal.

4.2 Core operators

The language is built around three core operators that determine how programs interact with memory:

```
Ident := exp    # Assign
Ident -> LVal   # Alias
LVal  = exp     # Write
```

These operators correspond respectively to allocation and binding, aliasing of existing locations, and mutation of stored values.

4.3 Environment-store model

The semantic model assumes an unbounded abstract store and does not model deallocation or garbage collection. Program state consists of an environment and a store, where the environment maps variables to locations and the store maps locations to values:

$$\mathit{Environment} : \mathit{Var} \to \mathit{Loc}$$

$$\mathit{Store} : \mathit{Loc} \to \mathit{Val}$$

The environment determines how names refer to locations and therefore captures aliasing relationships, while the store determines how primitive values and compound values are represented in memory. The language is designed so that many properties of mutable structures can be inferred directly from the syntax. The environment-store model is used to define how these effects interact with program state.

Each operator affects the program state in a different way:

- `:=` allocates exactly one fresh location for a value and binds the variable to that location. It is the only operator that affects both the environment and the store.

- `->` creates an alias by binding a variable to an existing location. It affects only the environment.

- `=` updates the value stored at a location. It affects only the store, and performs the update only if the new value is compatible with the existing value.

Since assign allocates a fresh location on every use, it establishes the initial type of the value stored at that location. Once created, a location should only be updated with compatible values. Both assign and alias can be used to rebind names, since neither can overwrite an existing value. In contrast, write updates the value stored at an existing location, and must therefore respect its type.

4.4 Arrays and records

Arrays and records are fixed-size compound values whose elements or fields are represented as locations in the store. In the final language, a compound value consists of an outer location containing an array or record value together with a collection of inner locations representing its elements or fields. These inner locations are allocated when the compound value is constructed. Since compound values contain locations, copying them may introduce sharing. The language therefore makes allocation of inner locations explicit using the notation `@n`, where `n` denotes the number of inner locations allocated during construction.

```
arr := @3[1,2,3]
rec := @2{data : 1, next : null}
```

The usefulness of this annotation relies on allocation being semantically uniform. Expressions allocate deterministically and independently of evaluation history, meaning that a construct allocates the same number of locations whenever it is evaluated. This property was originally introduced to preserve the design principle that operators should maintain stable semantic meaning independent of context. However, it also has the consequence of making sharing relationships inferable from the program text, since it allows allocation behavior to be determined from syntax.

4.5 Pointers

The language includes explicit pointers in the form of address values. For example, the expression `loc x` produces an address value corresponding to the location bound to `x`, while `deref(x)` follows that address value and accesses the referenced location. Pointers introduce an additional level of indirection by allowing locations to be referenced independently of the structures that contain them. This makes it possible to construct graph-shaped structures whose nodes preserve identity over time, rather than only tree-shaped structures where each node is referenced by a single parent. Consequently, the same logical data structure can be represented through multiple store topologies, including explicit pointer chains, shared nested structures and partially overlapping compound values.

4.6 Sharing and copying

Because allocation is explicit, sharing relationships can be reasoned about directly from the program text. If two compound values are constructed using distinct allocations, they cannot share state. Conversely, any sharing relationship must arise through explicit reuse of locations, either through aliasing, copying or pointers.

For example, in `arr := @2[1,3]; x := arr`, the first statement allocates two inner locations together with an outer location for `arr`, while the second statement allocates one fresh location for `x`. Since the second statement introduces only one new location

into the store when copying `arr`, it can be inferred that the two arrays must share their inner locations.

Making allocation explicit also makes it possible to distinguish syntactically between shallow copying and deep copying. Copying compound values preserves existing inner locations by default, while independent structures must be constructed explicitly through fresh allocation.

4.7 Abstraction level

The language aims to expose memory representation and memory effects without introducing the low-level complexity typically associated with systems programming languages. Therefore, it models only the allocation mechanisms necessary to express the semantic behavior relevant to the domain. Programmers do not manipulate raw addresses or perform manual memory management. Instead, memory behavior is exposed through a small set of constructs whose operational effects remain visible in the program text. The intention is that making these semantic effects explicit may support reasoning about mutable data structures without requiring an understanding of low-level machine-oriented memory mechanisms.

5

Formalization

This chapter defines the syntax, type system and operational semantics of the language. The operational semantics specify the behavior of programs, while the type system specifies a subset of programs that is considered valid in the language.

5.1 Syntax

The language is a small imperative core designed for reasoning about basic data structures, aliasing and mutation. The language intentionally omits higher-level features such as functions, lexical scope, and local declarations in order to keep the language minimal. Standard arithmetic and boolean constructs are included for completeness but are not central to the memory model.

The syntax is defined using a simplified BNF-style notation describing the abstract structure of programs.

$$\begin{aligned} \mathit{Expressions}: \quad & e ::= n \mid l \mid (e) \mid \text{true} \mid \text{false} \mid \text{null} \mid \text{loc}\, x \mid e + e \mid e - e \mid e * e \\ & \mid e == e \mid e != e \mid e < e \mid e <= e \mid e > e \mid e >= e \\ & \mid e \,\text{and}\, e \mid e \,\text{or}\, e \mid \text{not}\, e \mid @n[e_1, \ldots, e_n] \mid @n\{f_1 : e_1, \ldots, f_n : e_n\} \end{aligned}$$

$$\mathit{LValues}: \quad l ::= x \mid l[e] \mid l.f \mid \text{deref}(e)$$

$$\begin{aligned} \mathit{Statements}: \quad & s ::= \{s\} \mid \text{skip} \mid s; s \mid x := e \mid x \to l \mid l = e \\ & \mid \text{if}\, e \,\text{then}\, s \,\text{else}\, s \mid \text{while}\, e \,\text{do}\, s \end{aligned}$$

$$\mathit{Values}: \quad v ::= n \mid \text{true} \mid \text{false} \mid \text{null} \mid \text{addr}(\ell) \mid [\ell_1, \ldots, \ell_n] \mid \{f_1 : \ell_1, \ldots, f_n : \ell_n\}$$

In the grammar, l denotes the syntactic category of lvalues and should not be confused with the metavariable ℓ, which denotes store locations.

The language is defined by three main syntactic categories: expressions, lvalues, and statements. Expressions evaluate to values and may update the store through allocation. Lvalues evaluate to locations in the store and denote mutable memory positions. Statements describe computations that may update both the environment and the store.

Expressions include lvalues as a subcategory in the grammar. This allows variables, array accesses, field accesses, and dereferenced address values to be used wherever expressions are permitted. When an lvalue appears as an expression, evaluation implicitly reads the value stored at the referenced location.

Values describe the runtime objects produced during program execution. Primitive values include integers, booleans, and null. Compound values such as arrays and records contain locations, allowing sharing and aliasing to be visible through the store.

Arrays and records are annotated with a size parameter `@n`, which specifies the number of inner locations allocated during evaluation. The annotation must match the number of elements or fields in the corresponding construct. Making allocation explicit in the syntax exposes store behavior at the language level and allows sharing relationships and memory topology to be reasoned about from program text.

Address values are represented separately from lvalues, and are created by expressions such as $\mathrm{loc}\, x$, which constructs an explicit pointer that refers to the location currently bound to x. Since address values are values rather than store locations, expressions that evaluate to address values must be dereferenced explicitly using the construct $\mathrm{deref}(e)$. This separation between address values and lvalues allows the language to support both high-level variable, array, and record access and explicit reference manipulation.

The following metavariables are used throughout the formalization:

$$n \in \mathbb{Z} \quad b \in \{\mathrm{true}, \mathrm{false}\} \quad \ell \in \mathrm{Loc} \quad x \in \mathrm{Var} \quad f \in \mathrm{Field}$$

Loc denotes the set of store locations, Var the set of variable names, and Field the set of record field names.

The following metavariables are used to group families of operators:

$$\oplus \in \{+, -, *\} \qquad \odot \in \{<, >, \leq, \geq\} \qquad \equiv \in \{==, \neq\}$$

Arithmetic operators $\oplus$ operate on integers and produce integer values. Integer comparison operators $\odot$ compare integer values and produce booleans, and equality operators $\equiv$ compare values of the same type and produce booleans.

5.2 Type system

The type system is static and structural. Its primary goal is to preserve local consistency of memory operations by restricting mutation to compatible values. The type system does not include store typing, recursive types or full shape analysis. Recursive store structures are instead represented using finite unfoldings together with a restricted compatibility relation used for list-like encodings. This design keeps the system simple and closely aligned with the operational semantics of the language, where allocation, sharing, and mutation are observable through the store.

The set of types is defined as:

$$\tau ::= \text{Int} \mid \text{Bool} \mid \text{Null} \mid \text{Addr}\,\tau \mid \text{Array}(\tau, n) \mid \text{Record}(F)$$

The type constructor $\text{Addr}\,\tau$ classifies address values referring to locations containing values of type τ. Address values are first-class values in the language and are distinct from store locations, which are abstract components of the operational semantics. The type Null classifies the value null, which represents the absence of a valid reference target. The language treats Null as a separate type rather than merging it with address types. This keeps explicit addresses and absence-of-reference semantically distinct, simplifying reasoning about references and preventing null from implicitly behaving as a general pointer value. However, the compatibility relation used during mutation includes a restricted relaxation, allowing null terminators in recognized linked-list structures to be replaced by pointers to valid list nodes. Array types are written as $\text{Array}(\tau, n)$, where τ is the element type and n is the fixed array length. Record types are written as $\text{Record}(F)$, where F is a finite mapping from field names to types.

Typing judgments are defined relative to a type environment Γ, which maps variables to types. Three judgment forms are used:

$$\Gamma \vdash e : \tau \quad \Gamma \vdash l : \tau \quad \Gamma \vdash s \rightsquigarrow \Gamma'$$

The first judgment assigns types to expressions and the second assigns types to lvalues. For lvalues, the type τ denotes the type of the value stored at the referenced location rather than the type of the location itself. Statements do not evaluate to values, but may transform the typing environment. The judgment $\Gamma \vdash s \rightsquigarrow \Gamma'$ therefore describes how statement execution affects the typing environment.

The notation $\Gamma(x)$ denotes a lookup of variable x in Γ, while $\Gamma[x \mapsto \tau]$ denotes the typing environment obtained by extending Γ with the binding $x : \tau$, which replaces any previous binding for x if one exists.

5.2.1 Expression typing

Expression typing assigns types to expressions. Arithmetic operators require integer operands, logical operators require boolean operands, and compound values derive their types from their contained elements.

$$\textsc{T-Int}\ \frac{}{\Gamma \vdash n : \text{Int}} \qquad \textsc{T-Null}\ \frac{}{\Gamma \vdash \text{null} : \text{Null}} \qquad \textsc{T-Bool}\ \frac{}{\Gamma \vdash b : \text{Bool}}$$

$$\textsc{T-Paren}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e) : \tau} \qquad \textsc{T-Addr}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash \text{loc}\, x : \text{Addr}\, \tau} \qquad \textsc{T-Not}\ \frac{\Gamma \vdash e : \text{Bool}}{\Gamma \vdash \text{not}\, e : \text{Bool}}$$

$$\textsc{T-And}\ \frac{\Gamma \vdash e_1 : \text{Bool} \qquad \Gamma \vdash e_2 : \text{Bool}}{\Gamma \vdash e_1 \,\text{and}\, e_2 : \text{Bool}} \qquad \textsc{T-Or}\ \frac{\Gamma \vdash e_1 : \text{Bool} \qquad \Gamma \vdash e_2 : \text{Bool}}{\Gamma \vdash e_1 \,\text{or}\, e_2 : \text{Bool}}$$

$$\textsc{T-Arith}\ \frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 \oplus e_2 : \text{Int}} \qquad \textsc{T-IntComp}\ \frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 \odot e_2 : \text{Bool}}$$

$$\textsc{T-Eq}\ \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \equiv e_2 : \text{Bool}} \qquad \textsc{T-Array}\ \frac{\Gamma \vdash e_1 : \tau \qquad \cdots \qquad \Gamma \vdash e_n : \tau}{\Gamma \vdash @n[e_1, \ldots, e_n] : \text{Array}(\tau, n)}$$

$$\textsc{T-Record}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n \qquad f_1, \ldots, f_n \ \text{distinct}}{\Gamma \vdash @n\{f_1 : e_1, \ldots, f_n : e_n\} : \text{Record}(f_1 : \tau_1, \ldots, f_n : \tau_n)}$$

T-Int, T-Null, T-Bool, and T-Addr assign types to the primitive literal and address forms of the language. Their resulting types correspond to the type constructors introduced earlier in this chapter.

T-Array requires all array elements to have the same type. Array types include their statically known size as part of the type. Since array types are inferred from their elements, empty arrays are not included in the language.

T-Record assigns record types structurally from their fields. Record fields must be distinct, and the resulting record type is determined by the mapping from field names to field types.

5.2.2 LValue typing

LValue typing assigns types to lvalues, where the assigned type denotes the type of the value stored at the referenced location.

$$\text{T-Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{T-ArrIndex}\ \frac{\Gamma \vdash l : \text{Array}(\tau, n) \qquad \Gamma \vdash e : \text{Int}}{\Gamma \vdash l[e] : \tau}$$

$$\text{T-FieldAccess}\ \frac{\Gamma \vdash l : \text{Record}(F) \qquad F(f) = \tau}{\Gamma \vdash l.f : \tau} \qquad \text{T-Deref}\ \frac{\Gamma \vdash e : \text{Addr}\,\tau}{\Gamma \vdash \text{deref}(e) : \tau}$$

T-Var assigns variables the type associated with their current binding in the typing environment.

T-ArrIndex gives the type of array indexing. The left-hand side must denote an array value and the index expression must have type Int. The resulting lvalue has the element type of the array. The typing rule does not verify that the index lies within the array bounds. Bounds checking is performed dynamically by the interpreter.

T-FieldAccess gives the type of record field access. The notation $F(f) = \tau$ denotes a lookup of field f in the mapping F. Accessing a field f therefore produces an lvalue of type τ.

T-Deref assigns a type to explicit dereferencing. If an expression has type $\text{Addr}\,\tau$, dereferencing it produces an lvalue whose stored value has type τ. $\text{deref}(e)$ converts an address value into an lvalue, after which ordinary expression evaluation may read the contents of the referenced location.

5.2.3 Statement typing

The typing environment is dynamic and may evolve during type checking. In particular, variable introduction via the assign operator and alias creation may introduce new bindings or update the type associated with an existing variable.

$$\text{T-Assign}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash x := e \rightsquigarrow \Gamma[x \mapsto \tau]} \qquad \text{T-Alias}\ \frac{\Gamma \vdash l : \tau}{\Gamma \vdash x \to l \rightsquigarrow \Gamma[x \mapsto \tau]}$$

$$\text{T-Write}\ \frac{\Gamma \vdash l : \tau_1 \qquad \Gamma \vdash e : \tau_2 \qquad \text{Compatible}(\tau_1, \tau_2)}{\Gamma \vdash l = e \rightsquigarrow \Gamma} \qquad \text{T-Block}\ \frac{\Gamma \vdash s \rightsquigarrow \Gamma'}{\Gamma \vdash \{s\} \rightsquigarrow \Gamma'} \qquad \text{T-Skip}\ \frac{}{\Gamma \vdash \text{skip} \rightsquigarrow \Gamma}$$

$$\text{T-Seq}\ \frac{\Gamma \vdash s_1 \rightsquigarrow \Gamma' \qquad \Gamma' \vdash s_2 \rightsquigarrow \Gamma''}{\Gamma \vdash s_1; s_2 \rightsquigarrow \Gamma''} \qquad \text{T-If}\ \frac{\Gamma \vdash e : \text{Bool} \qquad \Gamma \vdash_{\text{safe}} s_1 \qquad \Gamma \vdash_{\text{safe}} s_2}{\Gamma \vdash \text{if}\ \ e\ \ \text{then}\ \ s_1\ \ \text{else}\ \ s_2 \rightsquigarrow \Gamma}$$

$$\text{T-While}\ \frac{\Gamma \vdash e : \text{Bool} \qquad \Gamma \vdash_{\text{safe}} s}{\Gamma \vdash \text{while}\ \ e\ \ \text{do}\ \ s \rightsquigarrow \Gamma}$$

T-Assign types the introduction of variables. The right-hand side expression is first typed as τ, after which the environment is updated with the binding $x \mapsto \tau$. Semantically, assignment in this language allocates a fresh location and binds the variable to it.

T-Alias types alias statements by giving the name the same type as the referenced lvalue. Unlike assignment, aliasing does not allocate a fresh location.

T-Write types mutation of existing locations. The left-hand side lvalue and right-hand side expression are typed independently. $\text{Compatible}(\tau_1, \tau_2)$ is then used to determine whether a mutation is permitted. Since mutation updates only existing store locations, the typing environment is not recomputed after mutation.

T-Block preserves the typing behavior of the enclosed statement. Blocks are included to allow multiple statements to appear within control-flow constructs such as `if` and `while`.

T-Seq threads the typing environment through sequential statements, where the environment produced by the first statement becomes the input environment of the second statement. This allows variable bindings established earlier in the program to be used in later statements.

T-If and T-While use the auxiliary judgment $\Gamma \vdash_{\text{safe}} s$, which restricts allocation and alias creation in conditional branches and loop bodies.

5.2.4 Auxiliary definitions

This section formalizes the auxiliary predicates and judgments referenced by the statement typing rules.

5.2.4.1 Compatibility predicates

The compatibility relation used by T-Write has two helper predicates that recognize tree-shaped and graph-shaped linked-list types. The predicate *IsTreeList* is defined inductively using a base case and a recursive case:

$$\text{IsTreeList}(\text{Record}\{\texttt{data} : \text{Int}, \texttt{next} : \text{Null}\})$$

$$\frac{\text{IsTreeList}(\tau)}{\text{IsTreeList}(\text{Record}\{\texttt{data} : \text{Int}, \texttt{next} : \tau\})}$$

The first rule says that a record with `data` of type Int and `next` of type Null is a tree-list node. The second rule says that a record with `data` of type Int and `next` of type τ is a tree-list node if τ is also a tree-list node. The predicate applies only to records containing exactly the fields `data` and `next`.

The predicate *IsGraphList* has the same structure but the recursive case goes through an address type:

$$\text{IsGraphList}(\text{Record}\{\texttt{data} : \text{Int}, \texttt{next} : \text{Null}\})$$

$$\frac{\text{IsGraphList}(\tau)}{\text{IsGraphList}(\text{Record}\{\texttt{data} : \text{Int}, \texttt{next} : \text{Addr}(\tau)\})}$$

The base case recognizes a terminal list node whose next field is Null. The recursive case recognizes a node whose `next` field contains an address to another valid list node.

5.2.4.2 Compatibility relation

The compatibility relation extends structural equality with a relaxation for validated list types.

$$\begin{aligned}
\text{Compatible}(\text{Addr}(\tau_1), \text{Addr}(\tau_2)) &\quad \text{if } \text{IsGraphList}(\tau_1) \wedge \text{IsGraphList}(\tau_2) \\
\text{Compatible}(\text{Addr}(\tau_1), \text{Addr}(\tau_2)) &\quad \text{if } \text{IsTreeList}(\tau_1) \wedge \text{IsTreeList}(\tau_2) \\
\text{Compatible}(\text{Null}, \text{Addr}(\tau)) &\quad \text{if } \text{IsGraphList}(\tau) \vee \text{IsTreeList}(\tau) \\
\text{Compatible}(\tau_1, \tau_2) &\quad \text{if } \tau_1 = \tau_2
\end{aligned}$$

The first two cases implement the shallow compatibility rule: two pointer types are considered compatible when both target types belong to the same linked-list family, regardless of unfolding depth. The third case permits null values to be replaced by pointers to validated list nodes during mutation. All other types require structural equality.

5.2.4.3 Relaxation

The relaxation is necessary to allow linked-list updates and pointer-based traversal. Compatible linked-list unfoldings are treated as equivalent during mutation even when their nesting depth differs. For example, the relaxation allows overwriting a value of type:

```
Addr(Record {data : Int,
             next : Record {data : Int,
                            next : Null}})
```

with a value of type:

```
Addr(Record {data : Int, next : Null})
```

which would not be allowed under strict equality.

The helper predicates recursively verify that the entire structure belongs to one of the permitted linked-list families before any relaxation is applied. In addition to checking field names and recursive shape, the helper predicates also ensure that each node in a list contains exactly two fields. The size restriction prevents unrelated record structures from accidentally satisfying the linked-list predicates through partial field overlap. A consequence of this restriction is that doubly linked lists are not covered by the relaxation, which is a limitation of the type system.

Once a structure has been recognized as a valid tree-shaped or graph-shaped list, the shallow compatibility rule is applied only to pointers to compatible list nodes. Only the outer pointer structure is compared (rather than the entire recursively unfolded type). This allows the type checker to preserve distinctions between null boundaries, pointer boundaries and embedded structure. Without these distinctions, recursively unfolded list types could collapse into a single broad compatibility class, reducing the type checker's ability to distinguish between different linked-list representations.

The relaxation has two purposes. First, it supports mutation of graph-shaped linked lists without requiring recursive types, which is necessary for standard linked-list operations and cursor-style traversal. Second, it permits null terminators in valid linked-list structures to be replaced by pointers to compatible list nodes, allowing linked lists to be extended dynamically. The reverse conversion from pointers to null is intentionally excluded. Allowing null terminators to be replaced by pointers

is conservative, since subsequent traversals are checked against the original finite unfolding and may therefore be rejected. The reverse conversion would be less conservative because traversals accepted by the type checker could encounter null values not reflected in the typing environment.

As a consequence of the relaxation, static type information may become less precise after updates. In particular, the type checker does not recompute recursive linked structure after updates involving null terminators. The environment therefore continues to describe the original finite unfolding even after a null terminator has been operationally replaced by a pointer to another list node. The effect is that programs that are operationally valid may later be rejected because the typing environment no longer fully reflects the runtime structure being traversed.

The shallow compatibility rule does not extend to records in general. Records containing additional fields, different field types, or incompatible recursive structure are rejected by the helper predicates and require ordinary structural equality under mutation.

5.2.4.4 Control-flow predicates

The auxiliary judgment $\Gamma \vdash_{\text{safe}} s$ is used by T-If and T-While to identify control-flow bodies that do not change the typing environment. A statement is safe if it satisfies the predicate *NoEffects* and type-checks without changing the environment. *NoEffects* recursively traverses statements and excludes operations that allocate fresh locations or introduce alias relationships.

$$\frac{\text{NoEffects}(s) \qquad \Gamma \vdash s \rightsquigarrow \Gamma}{\Gamma \vdash_{\text{safe}} s} \qquad \frac{\text{NoAllocExp}(e)}{\text{NoEffects}(l = e)} \qquad \frac{}{\text{NoEffects}(\text{skip})}$$

$$\frac{\text{NoEffects}(s_1) \qquad \text{NoEffects}(s_2)}{\text{NoEffects}(s_1; s_2)} \qquad \frac{\text{NoEffects}(s)}{\text{NoEffects}(\{s\})}$$

$$\frac{\text{NoEffects}(s_1) \qquad \text{NoEffects}(s_2)}{\text{NoEffects}(\text{if } e \text{ then } s_1 \text{ else } s_2)} \qquad \frac{\text{NoEffects}(s)}{\text{NoEffects}(\text{while } e \text{ do } s)}$$

No rule is provided for variable introduction or alias creation. Consequently, these operations are not permitted inside control-flow bodies.

Since allocation may occur within expressions, the auxiliary predicate *NoAllocExp* identifies expressions that do not allocate fresh locations.

$$\frac{}{\text{NoAllocExp}(n)} \qquad \frac{}{\text{NoAllocExp}(b)} \qquad \frac{}{\text{NoAllocExp}(\text{null})} \qquad \frac{}{\text{NoAllocExp}(l)}$$

$$\frac{}{\text{NoAllocExp}(\text{loc}\, x)} \qquad \frac{\text{NoAllocExp}(e)}{\text{NoAllocExp}(\text{not}\, e)} \qquad \frac{\text{NoAllocExp}(e)}{\text{NoAllocExp}((e))}$$

$$\frac{\text{NoAllocExp}(e_1) \qquad \text{NoAllocExp}(e_2)}{\text{NoAllocExp}(e_1 \oplus e_2)} \qquad \frac{\text{NoAllocExp}(e_1) \qquad \text{NoAllocExp}(e_2)}{\text{NoAllocExp}(e_1 \odot e_2)}$$

$$\frac{\text{NoAllocExp}(e_1) \qquad \text{NoAllocExp}(e_2)}{\text{NoAllocExp}(e_1 \equiv e_2)} \qquad \frac{\text{NoAllocExp}(e_1) \qquad \text{NoAllocExp}(e_2)}{\text{NoAllocExp}(e_1 \,\text{and}\, e_2)}$$

$$\frac{\text{NoAllocExp}(e_1) \qquad \text{NoAllocExp}(e_2)}{\text{NoAllocExp}(e_1 \,\text{or}\, e_2)}$$

No rules are provided for array or record expressions since they allocate fresh locations.

5.2.4.5 Control-flow restrictions

The typing environment may change across sequential statements as variables are introduced, rebound or aliased. This becomes difficult to reason about in the presence of repeated or branching execution, since loops and conditionals would otherwise require the type checker to track how environments and aliasing relationships evolve across multiple execution paths. To avoid this, allocation and alias creation are restricted inside `while` and `if` bodies through the auxiliary judgment $\Gamma \vdash_{\text{safe}} s$, allowing T-While and T-If to safely return the original environment Γ in their conclusions.

5.3 Operational semantics

The operational semantics is defined using big-step evaluation and assumes that programs have been type checked. The rules do not model runtime errors such as out-of-bounds array access or dereference of non-address values. These conditions are checked dynamically by the interpreter.

Programs execute relative to a runtime state consisting of an environment ρ and a store σ. Evaluation follows a call-by-value strategy, meaning that subexpressions are fully evaluated before their results are used. Evaluation may update the store through allocation and mutation, where allocation introduces fresh locations not already present in the current store. All variables exist within a single global mutable environment, allowing identifiers to be rebound during execution.

Three judgment forms are used:

$$\rho, \sigma \vdash_e e \Downarrow v, \sigma' \quad \rho, \sigma \vdash_l l \Downarrow \ell, \sigma' \quad \rho, \sigma \vdash_s s \Downarrow \rho', \sigma'$$

Expression evaluation produces a value and may update the store, but never the environment. LValue evaluation produces a location and may propagate store effects originating from subexpressions. Statement evaluation may update both the environment and the store.

5.3.1 Expression semantics

The interpretations of arithmetic, integer comparison, and equality operators are given by the following meta-functions:

$$[\![\oplus]\!] : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$
$$[\![\odot]\!] : \mathbb{Z} \times \mathbb{Z} \to \{\text{true}, \text{false}\}$$
$$[\![\equiv]\!] : \text{Value} \times \text{Value} \to \{\text{true}, \text{false}\}$$

The following rules define evaluation of expressions:

$$\textsc{Num}\ \frac{}{\rho, \sigma \vdash_e n \Downarrow n, \sigma} \qquad \textsc{Null}\ \frac{}{\rho, \sigma \vdash_e \text{null} \Downarrow \text{null}, \sigma} \qquad \textsc{True}\ \frac{}{\rho, \sigma \vdash_e \text{true} \Downarrow \text{true}, \sigma}$$

$$\textsc{False}\ \frac{}{\rho, \sigma \vdash_e \text{false} \Downarrow \text{false}, \sigma} \qquad \textsc{Paren}\ \frac{\rho, \sigma \vdash_e e \Downarrow v, \sigma'}{\rho, \sigma \vdash_e (e) \Downarrow v, \sigma'} \qquad \textsc{Address}\ \frac{\rho(x) = \ell}{\rho, \sigma \vdash_e \text{loc}\, x \Downarrow \text{addr}(\ell), \sigma}$$

$$\textsc{ELValue}\ \frac{\rho, \sigma \vdash_l l \Downarrow \ell, \sigma' \qquad \sigma'(\ell) = v}{\rho, \sigma \vdash_e l \Downarrow v, \sigma'} \qquad \textsc{Not}\ \frac{\rho, \sigma \vdash_e e \Downarrow b, \sigma'}{\rho, \sigma \vdash_e \text{not}\, e \Downarrow \neg b, \sigma'}$$

$$\textsc{And}\ \frac{\rho, \sigma \vdash_e e_1 \Downarrow b_1, \sigma' \qquad \rho, \sigma' \vdash_e e_2 \Downarrow b_2, \sigma''}{\rho, \sigma \vdash_e e_1 \,\text{and}\, e_2 \Downarrow (b_1 \wedge b_2), \sigma''} \qquad \textsc{Or}\ \frac{\rho, \sigma \vdash_e e_1 \Downarrow b_1, \sigma' \qquad \rho, \sigma' \vdash_e e_2 \Downarrow b_2, \sigma''}{\rho, \sigma \vdash_e e_1 \,\text{or}\, e_2 \Downarrow (b_1 \vee b_2), \sigma''}$$

$$\textsc{Arith}\ \frac{\rho, \sigma \vdash_e e_1 \Downarrow n_1, \sigma' \qquad \rho, \sigma' \vdash_e e_2 \Downarrow n_2, \sigma''}{\rho, \sigma \vdash_e e_1 \oplus e_2 \Downarrow [\![\oplus]\!](n_1, n_2), \sigma''} \qquad \textsc{IntComp}\ \frac{\rho, \sigma \vdash_e e_1 \Downarrow n_1, \sigma' \qquad \rho, \sigma' \vdash_e e_2 \Downarrow n_2, \sigma''}{\rho, \sigma \vdash_e e_1 \odot e_2 \Downarrow [\![\odot]\!](n_1, n_2), \sigma''}$$

$$\textsc{EqComp}\ \frac{\rho, \sigma \vdash_e e_1 \Downarrow v_1, \sigma' \qquad \rho, \sigma' \vdash_e e_2 \Downarrow v_2, \sigma''}{\rho, \sigma \vdash_e e_1 \equiv e_2 \Downarrow [\![\equiv]\!](v_1, v_2), \sigma''}$$

$$\textsc{Array}\ \frac{\rho, \sigma \vdash_e e_1 \Downarrow v_1, \sigma'\ \rho, \sigma' \vdash_e e_2 \Downarrow v_2, \sigma''\ \cdots\ \rho, \sigma^{(n-1)} \vdash_e e_n \Downarrow v_n, \sigma^{(n)}\ \ell_1, \ldots, \ell_n \,\text{fresh}}{\rho, \sigma \vdash_e @n[e_1, \ldots, e_n] \Downarrow [\ell_1, \ldots, \ell_n], \sigma^{(n)}[\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n]}$$

$$\textsc{Record}\ \frac{\rho, \sigma \vdash_e e_1 \Downarrow v_1, \sigma'\ \rho, \sigma' \vdash_e e_2 \Downarrow v_2, \sigma''\ \cdots\ \rho, \sigma^{(n-1)} \vdash_e e_n \Downarrow v_n, \sigma^{(n)}\ \ell_1, \ldots, \ell_n \,\text{fresh}}{\rho, \sigma \vdash_e @n\{f_1 : e_1, \ldots, f_n : e_n\} \Downarrow \{f_1 : \ell_1, \ldots, f_n : \ell_n\}, \sigma^{(n)}[\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n]}$$

The operational semantics distinguishes between locations and address values. Locations exist only inside the semantic model and are never manipulated directly by programs. Null is treated as an ordinary runtime value rather than a reserved memory location. Address values are ordinary runtime values that refer to locations.

Address constructs an address value by retrieving the location bound to a variable in the environment and wrapping it as a value of the form addr(ℓ).

ELValue evaluates an lvalue to a location and then reads the corresponding value from the store. This provides the implicit dereferencing behavior that allows lvalues to appear in the expression position.

Logical operators evaluate both operands unconditionally. In many languages, an expression of the form e_1 and e_2 will skip evaluating e_2 when e_1 already determines the result. This language avoids short-circuit evaluation in order to preserve explicit and context-independent operational semantics. Since expressions may contain observable effects such as allocation or dereferencing, short-circuit evaluation would make it impossible to determine syntactically whether a subexpression executes. By evaluating all operands unconditionally, the language preserves a direct correspondence between syntax and observable memory effects.

Array and record elements are evaluated sequentially from left to right with state threading. Each expression is evaluated in the current state, and the resulting state is passed to the next evaluation. After evaluation, fresh locations are allocated for the resulting values.

5.3.2 LValue semantics

LValues denote locations in the store. The following rules define their evaluation:

$$\textsc{Var}\quad \frac{\rho(x) = \ell}{\rho, \sigma \vdash_l x \Downarrow \ell, \sigma} \qquad \textsc{Deref}\quad \frac{\rho, \sigma \vdash_e e \Downarrow \mathrm{addr}(\ell), \sigma'}{\rho, \sigma \vdash_l \mathrm{deref}\, e \Downarrow \ell, \sigma'}$$

$$\textsc{ArrIndex}\quad \frac{\rho, \sigma \vdash_l l \Downarrow \ell, \sigma' \qquad \rho, \sigma' \vdash_e e \Downarrow i, \sigma'' \qquad \sigma''(\ell) = [\ell_0, \ldots, \ell_{n-1}] \qquad 0 \le i < n}{\rho, \sigma \vdash_l l[e] \Downarrow \ell_i, \sigma''}$$

$$\textsc{FieldAccess}\quad \frac{\rho, \sigma \vdash_l l \Downarrow \ell, \sigma' \qquad \sigma'(\ell) = \{f_1 : \ell_1, \ldots, f_n : \ell_n\} \qquad f = f_i}{\rho, \sigma \vdash_l l.f \Downarrow \ell_i, \sigma'}$$

Variables evaluate to the location currently bound to the variable in the environment.

Deref converts an address value into the location it refers to. This allows address values produced by Address to be used in contexts that require lvalues, such as mutation targets. Together, Deref, Address, and ELValue distinguish between store locations, address values referring to locations, and values stored at those locations.

Array indexing and field access operate by following locations stored inside arrays and records. For array indexing, the lvalue is first evaluated to an array location and the index expression is then evaluated. The lookup is performed after evaluation of the index expression, and therefore uses the most recently produced store. The order of these two premises does not affect the result for programs accepted by the type checker. Since the type checker requires the index expression to have type `Int`,

it cannot perform array or record allocation and therefore cannot modify the array being indexed.

5.3.3 Statement semantics

The following rules define the evaluation of statements:

$$\textsc{Assign}\ \frac{\rho, \sigma \vdash_e e \Downarrow v, \sigma' \qquad \ell \,\text{fresh}}{\rho, \sigma \vdash_s x := e \Downarrow \rho[x \mapsto \ell], \sigma'[\ell \mapsto v]} \qquad \textsc{Alias}\ \frac{\rho, \sigma \vdash_l l \Downarrow \ell, \sigma'}{\rho, \sigma \vdash_s x \to l \Downarrow \rho[x \mapsto \ell], \sigma'}$$

$$\textsc{Write}\ \frac{\rho, \sigma \vdash_l l \Downarrow \ell, \sigma' \qquad \rho, \sigma' \vdash_e e \Downarrow v, \sigma''}{\rho, \sigma \vdash_s l = e \Downarrow \rho, \sigma''[\ell \mapsto v]} \qquad \textsc{Block}\ \frac{\rho, \sigma \vdash_s s \Downarrow \rho', \sigma'}{\rho, \sigma \vdash_s \{s\} \Downarrow \rho', \sigma'}$$

$$\textsc{Skip}\ \frac{}{\rho, \sigma \vdash_s \text{skip} \Downarrow \rho, \sigma} \qquad \textsc{Seq}\ \frac{\rho, \sigma \vdash_s s_1 \Downarrow \rho', \sigma' \qquad \rho', \sigma' \vdash_s s_2 \Downarrow \rho'', \sigma''}{\rho, \sigma \vdash_s s_1; s_2 \Downarrow \rho'', \sigma''}$$

$$\textsc{If-False}\ \frac{\rho, \sigma \vdash_e e \Downarrow \text{false}, \sigma' \qquad \rho, \sigma' \vdash_s s_2 \Downarrow \rho'', \sigma''}{\rho, \sigma \vdash_s \text{if } e \text{ then } s_1 \text{ else } s_2 \Downarrow \rho'', \sigma''}$$

$$\textsc{If-True}\ \frac{\rho, \sigma \vdash_e e \Downarrow \text{true}, \sigma' \qquad \rho, \sigma' \vdash_s s_1 \Downarrow \rho'', \sigma''}{\rho, \sigma \vdash_s \text{if } e \text{ then } s_1 \text{ else } s_2 \Downarrow \rho'', \sigma''} \qquad \textsc{While-False}\ \frac{\rho, \sigma \vdash_e e \Downarrow \text{false}, \sigma'}{\rho, \sigma \vdash_s \text{while } e \text{ do } s \Downarrow \rho, \sigma'}$$

$$\textsc{While-True}\ \frac{\rho, \sigma \vdash_e e \Downarrow \text{true}, \sigma' \qquad \rho, \sigma' \vdash_s s \Downarrow \rho'', \sigma'' \qquad \rho'', \sigma'' \vdash_s \text{while } e \text{ do } s \Downarrow \rho''', \sigma'''}{\rho, \sigma \vdash_s \text{while } e \text{ do } s \Downarrow \rho''', \sigma'''}$$

The three core state-transforming operations of the language are assign, alias, and write.

Assign evaluates the right-hand-side expression, allocates a fresh location for the resulting value, and updates the environment to bind the variable to that location. Rebinding an existing variable therefore creates a new location rather than overwriting the previous one.

Alias evaluates the right-hand-side lvalue to an existing store location and binds the variable to that location without allocating new storage. Multiple variables may therefore refer to the same location.

Write performs mutation. The left-hand-side lvalue is evaluated to a target location and the right-hand-side expression is evaluated to a value, after which the store

is updated at the target location. Unlike Assign and Alias, Write updates an existing store location without modifying the environment.

Block introduces syntactic grouping of statements without creating a new scope or environment frame. Evaluating a block therefore has the same effect as evaluating the enclosed statement.

Seq evaluates statements in order by passing the resulting environment and store of the first statement to the second. Since allocation, rebinding, and mutation are observable effects, evaluation order is semantically significant.

If and While behave as in standard imperative languages. Conditional statements evaluate the guard expression and execute exactly one branch depending on the resulting boolean value. While-loops repeatedly evaluate the guard expression and execute the loop body as long as the condition is true. The semantics are defined recursively: after each iteration, evaluation continues from the resulting program state. Non-termination is represented by the absence of a finite derivation.

6

Runtime

An interpreter for the language has been implemented in Haskell, corresponding to the operational semantics defined in the previous chapter. The interpreter is used to generate the environment-store outputs shown throughout this chapter. Values are represented using Haskell data constructors such as `IntVal`, `BoolVal`, `RecVal`, and `ArrVal`, corresponding to the semantic values defined in the formalization. Store locations such as `Loc 0` are semantic objects used internally by the operational semantics and are never written by the programmer.

The core language constructs are shown through a series of concrete programs. Each example illustrates how a particular language construct, or combination of language constructs, affects the environment and store. The purpose of the examples is to demonstrate that the design exposes semantic behavior that is often implicit in standard high-level programming languages.

6.1 Assignment

The assign operator allocates a fresh location in the store, stores the value of the right-hand side expression in that location, and binds the left-hand side variable to the newly allocated location. The operator is also used to copy existing values. To keep its behavior uniform, `:=` always copies the value produced by the right-hand side expression and never reuses the location associated with that expression. The same evaluation rule therefore applies regardless of whether the expression evaluates to a primitive value, a compound value, or an existing variable. Although assignment never creates aliases, copying compound values may introduce sharing through their inner locations.

The simplest case is assignment of a primitive value:

```
x := 1;
y := x
```

Here, `x` allocates a location for the value 1, while `y` allocates a second location containing a copy of that value.

```
Environment (variable ↦ location):
x ↦ Loc 0
y ↦ Loc 1

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ IntVal 1
```

Both variables are bound to different locations in the store. Since the copied value does not contain inner locations, no sharing occurs.

Compound values preserve the same assignment semantics. Copying an array copies the outer structure while reusing references to existing inner locations:

```
arr := @3[1,2,3];
z := arr
```

```
Environment (variable ↦ location):
arr ↦ Loc 3
z   ↦ Loc 4

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ IntVal 2
Loc 2 ↦ IntVal 3
Loc 3 ↦ ArrVal [Loc 0, Loc 1, Loc 2]
Loc 4 ↦ ArrVal [Loc 0, Loc 1, Loc 2]
```

The assignment to `z` allocates exactly one new outer location. Since the copied array already contains references to its inner locations, the new array shares those locations. This means that sharing relationships can be inferred from the allocation behavior of the program. In this example, the occurrences of `:=` and `@3` reveal that only one additional location is introduced when `arr` is copied, making sharing unavoidable. This is a shallow copy: the outer array value is copied, while inner locations are shared. To construct an independent array, the inner locations must be allocated explicitly, either through direct construction or through an explicit construction that duplicates inner structure.

Rebinding a name using `:=` always allocates a fresh location and rebinds the name to this newly allocated location. The previously referenced location remains in the store, but may become unreachable from the environment if no other variable, pointer or structure refers to it.

For example, `x := 1; x := 2` allocates two distinct locations. After the second assignment, the original location containing `1` is no longer reachable from `x`. However,

previously allocated locations may remain reachable through pointers, aliases or shared structure. This makes it possible to construct linked structures incrementally:

```
rec := @1{field : null};
rec := @1{field : loc rec}
```

The second assignment allocates a fresh record and rebinds `rec`, but the new record contains a pointer to the previously allocated record. The old location therefore remains reachable through the newly constructed structure.

```
Environment (variable ↦ location):
rec ↦ Loc 3

Store (location ↦ value):
Loc 0 ↦ NullVal
Loc 1 ↦ RecVal { field : Loc 0 }
Loc 2 ↦ AddrVal Loc 1
Loc 3 ↦ RecVal { field : Loc 2 }
```

The assign operator can also be used to create pointers. Pointers are constructed using `:=` together with the `loc` construct, which produces the location of an existing variable:

```
x := true;
p := loc x
```

```
Environment (variable ↦ location):
x ↦ Loc 0
p ↦ Loc 1

Store (location ↦ value):
Loc 0 ↦ BoolVal True
Loc 1 ↦ AddrVal Loc 0
```

The pointer itself is stored as a value. The store location `Loc 0` therefore remains distinct from the address value `AddrVal Loc 0`. Pointers are intentionally restricted to identifiers to keep the language high-level. To mutate the referenced location through `p`, the address value must first be dereferenced explicitly.

6.2 Aliasing

The language distinguishes between aliasing and structural sharing. Aliasing is introduced explicitly through `->`, which binds a variable to an existing location. This makes alias relationships visible in the environment and straightforward to reason about from program structure. Sharing instead emerges from copying compound values containing inner locations. In this case, existing locations are reused without introducing new environment bindings. Although sharing relationships can be inferred from syntax since allocation is explicit in program text, reasoning about their effects still requires following how shared locations are reached through copied structures. The distinction therefore separates deliberate access paths introduced by the programmer from structural reuse arising during evaluation.

Multiple variables can refer to the same underlying structure. This makes it possible to introduce multiple access paths to the same mutable structure without copying:

```
arr := @2[1,2];
x -> arr;
y -> arr
```

Since aliasing is introduced only through `->`, alias relationships are visible in the program text. Here, `x`, `y` and `arr` all refer to the same array location, while the array itself is allocated only once:

```
Environment (variable ↦ location):
arr ↦ Loc 2
x ↦ Loc 2
y ↦ Loc 2

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ IntVal 2
Loc 2 ↦ ArrVal [Loc 0, Loc 1]
```

Like the assign operator, the alias operator may also rebind existing names:

```
arr := @1[1];
x -> arr;
y -> arr;
z := 2;
x -> z     # x is rebound
```

A previous aliasing relationship is always broken when `->` is used with an already existing name on the left-hand side. In this example, rebinding `x` changes its association from `arr` to `z`, while the relationship between `arr` and `y` remains unchanged:

```
Environment (variable ↦ location):
arr ↦ Loc 1
x ↦ Loc 2
y ↦ Loc 1
z ↦ Loc 2

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ ArrVal [Loc 0]
Loc 2 ↦ IntVal 2
```

Because variables are simply names referring to locations, both `->` and `:=` may be used to rebind any name to a different location, regardless of how the name was first introduced.

Unlike pointers, which are restricted to variables, aliases may target any expression that evaluates to a location, including array elements, record fields and dereferenced pointers. Since arrays and records contain inner locations, aliasing may therefore occur both between variables and within compound values.

For example:

```
arr := @2[true, false];
x -> arr[1]
```

binds `x` to the location of the second array element:

```
Environment (variable ↦ location):
arr ↦ Loc 2
x ↦ Loc 1

Store (location ↦ value):
Loc 0 ↦ BoolVal True
Loc 1 ↦ BoolVal False
Loc 2 ↦ ArrVal [Loc 0, Loc 1]
```

`x` can now be used to update the corresponding array position. This introduces a direct path to an inner location, allowing the element to be updated independently of the surrounding structure.

Field aliasing behaves similarly by allowing individual fields to be referenced directly through environment bindings:

```
rec := @2{data : 7, next : null};
x -> rec.data
```

```
Environment (variable ↦ location):
rec ↦ Loc 2
x ↦ Loc 0

Store (location ↦ value):
Loc 0 ↦ IntVal 7
Loc 1 ↦ NullVal
Loc 2 ↦ RecVal { data : Loc 0, next : Loc 1 }
```

Aliases can be used to traverse linked structures by repeatedly rebinding the same variable to `next` fields. Alias traversal therefore occurs through rebinding environment names to existing locations, rather than through repeated construction and dereferencing of pointer values.

6.3 Mutation

Mutation preserves the environment, updating only the contents of existing locations. Variables must therefore already be bound to locations using assign (`:=`) or alias (`->`) before the write operator can be used for updates. The language supports both cell-level mutation and mutation of compound values. Mutation is intended to preserve the operational structure of locations by allowing overwrites only with compatible values. For example, an array of size 2 cannot be overwritten with an array of size 3, although certain recursive structures such as linked lists are treated more permissively.

The first form of mutation, cell-level mutation, updates locations containing primitive values. Consider the evaluation of the following program:

```
x := @2[1,2];
z -> x
```

```
Environment (variable ↦ location):
x ↦ Loc 2
z ↦ Loc 2

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ IntVal 2
Loc 2 ↦ ArrVal [Loc 0, Loc 1]
```

The array expression is evaluated first, producing two integer values. Fresh locations are allocated in the store for these values, and the outer array structure is then allocated to contain references to those locations. The result is stored at a fresh location, and `x` is bound to that location in the environment. Next, `z -> x` introduces a new environment binding that refers to the same location as `x`. As a result, both `x`

and `z` refer to the same array value in the store.

The array stored at Loc 2 contains references to the inner locations created during evaluation. Both `x` and `z` may now be used to update this array:

```
x[0] = 4;
z[1] = 5
```

```
Environment (variable ↦ location):
x ↦ Loc 2
z ↦ Loc 2

Store (location ↦ value):
Loc 0 ↦ IntVal 4
Loc 1 ↦ IntVal 5
Loc 2 ↦ ArrVal [Loc 0, Loc 1]
```

Each update modifies one of the inner locations referenced by the array, while the array structure itself remains unchanged. This shows that cell-level mutation preserves existing sharing relationships, only modifying the targeted locations.

Mutation may also overwrite compound values. Such overwrites can both introduce new sharing relationships and break existing ones. The language supports two distinct forms of compound value overwrite, which produce different sharing behavior.

Consider the following example, where sharing is first introduced by `x` copying `arr1`:

```
arr1 := @2[1,2];
arr2 := @2[3,4];
x := arr1
```

```
Environment (variable ↦ location):
arr1 ↦ Loc 2
arr2 ↦ Loc 5
x ↦ Loc 6

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ IntVal 2
Loc 2 ↦ ArrVal [Loc 0, Loc 1]    # arr1
Loc 3 ↦ IntVal 3
Loc 4 ↦ IntVal 4
Loc 5 ↦ ArrVal [Loc 3, Loc 4]    # arr2
Loc 6 ↦ ArrVal [Loc 0, Loc 1]    # x
```

When `x` copies `arr1`, a shallow copy is produced, causing `x` and `arr1` to share inner locations. Consider what happens when the above program is extended with:

```
x = arr2
```

Since this mutation does not allocate new locations, the only possible effect is that the references inside the array value at Loc 6 are overwritten with the references stored inside `arr2`:

```
Loc 6 ↦ ArrVal [Loc 3, Loc 4]
```

The overwrite therefore breaks the sharing relationship between `x` and `arr1`, while simultaneously introducing sharing between `x` and `arr2`. Since `=` does not allocate a new outer location for `x`, the mutation updates only the internal references of the existing compound value.

A similar example demonstrates how shared mutation may propagate through several variables simultaneously:

```
arr1 := @2[1,2];
arr2 := arr1;
x := arr2;
x[0] = 5
```

```
Environment (variable ↦ location):
arr1 ↦ Loc 2
arr2 ↦ Loc 3
x ↦ Loc 4

Store (location ↦ value):
Loc 0 ↦ IntVal 5
Loc 1 ↦ IntVal 2
Loc 2 ↦ ArrVal [Loc 0, Loc 1]
Loc 3 ↦ ArrVal [Loc 0, Loc 1]
Loc 4 ↦ ArrVal [Loc 0, Loc 1]
```

It is visible from the program text that the only allocation of an array structure occurs in the expression `arr1 := @2[1,2]`, which introduces two inner locations and a single outer location bound to `arr1`. The subsequent bindings, `arr2 := arr1` and `x := arr2`, each allocate new outer locations, but do not allocate new inner locations. Because allocation of inner structure only happens in the first statement, all three variables refer to the same inner locations. As a result, mutation through one variable is reflected in all others, since they share the same underlying inner structure.

This demonstrates that although copying and mutation may introduce or break sharing relationships, sharing is ultimately determined by allocation behavior. Since all allocation is made explicit through `@n`, sharing can be inferred from the program text without inspecting the store.

A different form of mutation occurs when fresh inner locations are allocated during the overwrite of a compound value. Consider:

```
arr1 := @2[1,2];
x := arr1
```

This program results in a shallow copy where `x` and `arr1` share inner locations:

```
Environment (variable ↦ location):
arr1 ↦ Loc 2
x ↦ Loc 3

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ IntVal 2
Loc 2 ↦ ArrVal [Loc 0, Loc 1]
Loc 3 ↦ ArrVal [Loc 0, Loc 1]
```

Overwriting `x` with:

```
x = @2[3,4]
```

allocates two fresh inner locations during the construction of the array:

```
Environment (variable ↦ location):
arr1 ↦ Loc 2
x ↦ Loc 3

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ IntVal 2
Loc 2 ↦ ArrVal [Loc 0, Loc 1]
Loc 3 ↦ ArrVal [Loc 4, Loc 5]
Loc 4 ↦ IntVal 3
Loc 5 ↦ IntVal 4
```

The environment confirms that `x` remains bound to Loc 3. However, the references inside the array value stored at that location now refer to newly allocated inner locations. The previous sharing relationship with `arr1` is therefore broken completely. The distinction between these two forms of compound overwrite is operationally significant. Non-allocating overwrite redirects sharing relationships toward already

existing locations, while allocation in combination with mutation replaces sharing relationships with newly allocated structures. The latter makes it possible to construct independent copies of compound values, which will be demonstrated in section 7.1.

Mutation may also occur through explicit pointers:

```
x := true;
p := loc x;
```

```
Environment (variable ↦ location):
p ↦ Loc 1
x ↦ Loc 0

Store (location ↦ value):
Loc 0 ↦ BoolVal True
Loc 1 ↦ AddrVal Loc 0
```

The pointer `p` stores the location of `x` as a value. To mutate the referenced location, the pointer value must first be dereferenced:

```
deref(p) = false
```

```
Environment (variable ↦ location):
p ↦ Loc 1
x ↦ Loc 0

Store (location ↦ value):
Loc 0 ↦ BoolVal False
Loc 1 ↦ AddrVal Loc 0
```

6.4 Pointers

Pointers are represented explicitly as address values. Unlike aliases, which operate through environment bindings, address values may be copied, stored inside compound values, and passed between locations. This introduces an additional level of indirection, making it possible to distinguish between updating values and updating references to values. Pointers are intentionally restricted compared to low-level languages such as C. There is no pointer arithmetic, manual memory management or construction of arbitrary addresses. Address values can only be created from existing identifiers using `loc`, and can only be used through controlled operations such as dereferencing. The additional level of indirection becomes visible operationally as extra store lookups.

A value can either be duplicated by copying it into new locations, or reused by

storing an address value in an existing location.

Consider the following two programs:

```
# Program 1
v1 := 1;
v2 := 2;
arr := @2[v1, v2]

# Program 2
v1 := 1;
v2 := 2;
arr := @2[loc v1, loc v2]
```

Although the programs appear structurally similar, they construct fundamentally different runtime representations.

In program 1, the values of `v1` and `v2` are copied into fresh array locations:

```
Environment (variable ↦ location):
arr ↦ Loc 4
v1 ↦ Loc 0
v2 ↦ Loc 1

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ IntVal 2
Loc 2 ↦ IntVal 1
Loc 3 ↦ IntVal 2
Loc 4 ↦ ArrVal [Loc 2, Loc 3]
```

The array therefore contains independent copies rather than references to the original variables. Mutating `v1` or `v2` will not affect the array, since no sharing exists between the variables and the array contents.

In programming languages where copying behaves differently depending on the type of the right-hand-side value, the meaning of program 1 is ambiguous. Copying a value into an array could mean either copying the value itself into the array, or copying a reference to the value into the array. In this language, explicit allocation makes it clear that the values of `v1` and `v2` are copied into the array.

To swap the positions of the values stored in the array in program 1, the array cells themselves must be overwritten:

```
arr[0] = v2;
arr[1] = v1
```

resulting in:

```
Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ IntVal 2
Loc 2 ↦ IntVal 2
Loc 3 ↦ IntVal 1
Loc 4 ↦ ArrVal [Loc 2, Loc 3]
```

Program 2 instead stores address values inside the array:

```
Environment (variable ↦ location):
arr ↦ Loc 4
v1 ↦ Loc 0
v2 ↦ Loc 1

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ IntVal 2
Loc 2 ↦ AddrVal (Loc 0)
Loc 3 ↦ AddrVal (Loc 1)
Loc 4 ↦ ArrVal [Loc 2,Loc 3]
```

The array now refers indirectly to `v1` and `v2` through stored pointers. This introduces two distinct ways to perform a swap.

The first approach rewires the references stored inside the array itself:

```
arr[0] = loc v2;
arr[1] = loc v1
```

which produces:

```
Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ IntVal 2
Loc 2 ↦ AddrVal (Loc 1)
Loc 3 ↦ AddrVal (Loc 0)
Loc 4 ↦ ArrVal [Loc 2,Loc 3]
```

Here, the underlying integer values remain unchanged. Only the address values at Loc 2 and Loc 3 are updated.

The second approach preserves the references but mutates the values stored at the referenced locations using a temporary variable:

```
temp := v1;
v1 = v2;
v2 = temp
```

resulting in:

```
Environment (variable ↦ location):
arr ↦ Loc 4
temp ↦ Loc 5
v1 ↦ Loc 0
v2 ↦ Loc 1

Store (location ↦ value):
Loc 0 ↦ IntVal 2
Loc 1 ↦ IntVal 1
Loc 2 ↦ AddrVal (Loc 0)
Loc 3 ↦ AddrVal (Loc 1)
Loc 4 ↦ ArrVal [Loc 2,Loc 3]
Loc 5 ↦ IntVal 1
```

In this case, the address values inside the array are unchanged, but the locations they refer to have been mutated. Since the array still contains the same inner locations, it observes the updated values automatically. The distinction between these two forms of updates becomes visible only because pointers introduce an explicit level of indirection. The runtime model therefore separates updating references from updating the values referenced by them.

The mutation can also be expressed through dereferencing the pointers stored inside the array:

```
temp := v1;
deref(arr[0]) = v2;
deref(arr[1]) = temp
```

which produces the same final store. Here, `arr[0]` and `arr[1]` evaluate to pointer values, and `deref` follows these address values to access the referenced locations before mutation occurs. Explicit address values therefore make it possible to construct shared reference-based structures while keeping the underlying indirection visible in the program.

7

Results

The following examples build on the operational model presented in the previous chapter and demonstrate how the language behaves when applied to several of the data structures and operations identified during the domain analysis. The first part examines how shallow and deep copying behave in the language. The second part examines three linked-list construction patterns, showing how they produce different memory topologies, sharing behaviors and traversal styles. The third part presents examples that expose limitations of the structural type checker when reasoning about mutable pointer-based structures and shows how those limitations motivated the design of the final compatibility relation. A final section demonstrates that the language can express standard data-structure manipulations and algorithms.

Linked lists in this chapter are sometimes displayed using a derived pretty-printed representation rather than raw store output. The pretty-printer follows links and nested records to reconstruct a readable list view, since larger structures may become difficult to inspect from the store representation.

7.1 Shallow and deep copying

The allocation behavior of the language makes the difference between shallow and deep copying visible in the syntax. A shallow copy can be expressed through ordinary use of the assign operator:

```
# Shallow copy
arr := @2[1,2];
copy := arr
```

Here, `copy` receives a fresh outer location, while the inner locations of the array are reused. The two arrays therefore become distinct outer structures that share the same inner locations. Mutations to shared elements will therefore be seen by both arrays.

The resulting store shows the sharing between `arr` and `copy`:

```
Environment (variable ↦ location):
arr  ↦ Loc 2
copy ↦ Loc 3

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ IntVal 2
Loc 2 ↦ ArrVal [Loc 0, Loc 1]
Loc 3 ↦ ArrVal [Loc 0, Loc 1]
```

Deep copying behaves differently. Instead of reusing existing inner locations, the program constructs a new structure using values read from the original structure. This means that mutations of one value cannot affect the internal state of the other. Deep copying can be expressed in the language by combining allocation and access operations:

```
# Deep copy
arr := @2[1,2];
copy := @2[arr[0], arr[1]]
```

This produces a deep copy because the expressions `arr[0]` and `arr[1]` evaluate to the values stored in the original array, while the array constructor allocates fresh inner locations for the elements of the new array.

The resulting store shows that both arrays are independent:

```
Environment (variable ↦ location):
arr ↦ Loc 2
copy ↦ Loc 5

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ IntVal 2
Loc 2 ↦ ArrVal [Loc 0, Loc 1]
Loc 3 ↦ IntVal 1
Loc 4 ↦ IntVal 2
Loc 5 ↦ ArrVal [Loc 3, Loc 4]
```

The language makes the cost difference between shallow and deep copying more visible. A shallow copy always allocates exactly one new outer location regardless of structure size, while a deep copy of an array with n elements requires allocating $n + 1$ locations and performing n element accesses. Both the time and space cost of deep copying therefore become visible in the program through repeated accesses and explicit allocation.

A deep copy of a record can be constructed similarly:

```
# Deep copy
rec := @2{data : 1, next : null};
copy := @2{data : rec.data, next : rec.next}
```

```
Environment (variable ↦ location):
copy ↦ Loc 5
rec ↦ Loc 2

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ NullVal
Loc 2 ↦ RecVal { data : Loc 0, next : Loc 1 }
Loc 3 ↦ IntVal 1
Loc 4 ↦ NullVal
Loc 5 ↦ RecVal { data : Loc 3, next : Loc 4 }
```

When array elements are address values rather than primitive values such as integers, copying behavior interacts with the pointer structure. Copying an address value preserves the reference it contains, while dereferencing before copying reconstructs the value at the referenced location into fresh storage. The distinction becomes visible in the following example:

```
# Deep copy
v := 1;
arr := @1[loc v];
copy := @1[deref(arr[0])]
```

```
Environment (variable ↦ location):
arr  ↦ Loc 2
copy ↦ Loc 4
v    ↦ Loc 0

Store (location ↦ value):
Loc 0 ↦ IntVal 1
Loc 1 ↦ AddrVal Loc 0
Loc 2 ↦ ArrVal [Loc 1]
Loc 3 ↦ IntVal 1
Loc 4 ↦ ArrVal [Loc 3]
```

The expression `arr[0]` evaluates to an address value, while `deref(arr[0])` follows that address and retrieves the value stored at the referenced location.

After mutating `v`:

```
v = 2
```

```
User-level (variable ↦ value):
arr  ↦ [Loc 0]
copy ↦ [1]
v    ↦ 2
```

Mutating `v` updates Loc 0, which is still referenced by the address value inside `arr`, so `arr` observes the change. Since `copy` holds an independent integer value it is unaffected by the update.

This example demonstrates that deep copying in the language is controlled explicitly through evaluation. Copying an address value preserves the referenced location, while dereferencing an address value reconstructs the pointed-to value into fresh storage. The boundary between shared and reconstructed structure is therefore directly visible in program syntax rather than hidden behind implicit runtime behavior.

7.2 Linked lists

Linked lists can be constructed in three distinct ways, where each construction pattern produces a different representation in the store and encourages a different traversal discipline. The representations differ in how nodes are allocated, and whether structure is preserved through identity, reconstructed through copying, or embedded through nesting. Graph-shaped construction preserves node identity through explicit addresses. Embedded construction allocates the entire structure in a single expression without reconstruction. Hybrid construction rebuilds the list by copying previously evaluated nodes.

7.2.1 Graph-shaped linked lists

A traditional imperative-style linked list is constructed using pointers:

```
# Graph-shaped linked list
node := @2{data : 3, next : null};
node := @2{data : 2, next : loc node};
node := @2{data : 1, next : loc node};
```

This is a referential construction pattern where each node is independently allocated, meaning that every node preserves its identity and `next` fields just store addresses to other nodes.

This construction pattern produces a graph-shaped structure in the store:

```
Environment (variable ↦ location):
node ↦ Loc 8

Store (location ↦ value):
Loc 0 ↦ IntVal 3
Loc 1 ↦ NullVal
Loc 2 ↦ RecVal { data : Loc 0, next : Loc 1 }
Loc 3 ↦ IntVal 2
Loc 4 ↦ AddrVal Loc 2
Loc 5 ↦ RecVal { data : Loc 3, next : Loc 4 }
Loc 6 ↦ IntVal 1
Loc 7 ↦ AddrVal Loc 5
Loc 8 ↦ RecVal { data : Loc 6, next : Loc 7 }
```

The store output reflects the syntax: `loc node` makes it clear `next` fields contain links to nodes instead of the nodes themselves. Since structures contain locations rather than values, the addresses to nodes are stored in individual locations, which means that traversal and updates become about mutating links rather than nodes. The explicit construct for pointers makes the program representation correspond closely to the store representation of the list.

Traversal follows the graph structure in the store, using a cursor-based traversal discipline.

```
curr :=  loc node;
curr = deref(curr).next
```

Traversal proceeds by first creating a pointer to the head of the list, and then mutating this pointer cursor in place. This is done by repeatedly dereferencing the current address to get the next node, and following the `next` field of this node to get the next address in the chain, which overwrites the value stored at the location bound to `curr`. Since traversal of graph-shaped linked lists only mutates the cursor itself, the nodes in the list are never copied or overwritten during traversal. As a result, advancing through the structure runs in constant time per step and preserves node identity throughout the traversal.

Insertion and deletion operations in graph-shaped lists are shown later in the expressiveness section. Since these operations primarily demonstrate that standard linked-list algorithms can be implemented within the language, they are less relevant to this section, which focuses on how different construction methods produce different memory topologies and traversal behaviors.

7.2.2 Tree-shaped linked lists

Linked lists can also be constructed through nesting nodes:

```
# Tree-shaped linked list
node := @2{data : 1, next :
          @2{data : 2, next :
            @2{data : 3, next :
               null}}};
```

This is an embedded construction pattern, where each child node is allocated within the construction of its parent. Unlike the graph-shaped and hybrid constructions, which build lists by repeatedly adding new nodes at the head of an existing list, nested construction creates the entire structure in a single expression. The resulting structure is built from the head toward the tail in forward order. The store shows a tree-shaped structure:

```
Environment (variable ↦ location):
node ↦ Loc 6

Store (location ↦ value):
Loc 0 ↦ IntVal 3
Loc 1 ↦ NullVal
Loc 2 ↦ IntVal 2
Loc 3 ↦ RecVal { data : Loc 0, next : Loc 1 }
Loc 4 ↦ IntVal 1
Loc 5 ↦ RecVal { data : Loc 2, next : Loc 3 }
Loc 6 ↦ RecVal { data : Loc 4, next : Loc 5 }
```

Traversal of the tree-shaped linked list is naturally structural rather than cursor-oriented:

```
# Traverse to the end of the list
curr -> node.next.next;
```

The traversal depth matches the depth of the target node in the list. Since `->` creates an alias rather than a copy, traversal requires no store mutation. The variable `curr` is simply rebound to the location reached by the path expression, while the store remains unchanged.

7.2.3 Hybrid linked lists

Linked structures can be constructed without explicit pointers or embedding nodes.

```
# Hybrid linked list
node := @2{data : 3, next : null};
node := @2{data : 2, next : node};
node := @2{data : 1, next : node};
```

This construction pattern combines reconstruction with referential sharing. Since records allocate fresh inner locations during construction, `next : node` produces a copy of the previous node, rather than a link to it. Because the values being copied are compound values containing locations, the inner locations are reused, producing shallow copies in the store. The hybrid list therefore combines structural duplication at the outer level with shared inner locations at the field level. The resulting semantics are neither strictly value-based nor strictly reference-based, but a hybrid of value reconstruction and imperative sharing.

The store makes the resulting sharing relationships easier to observe:

```
Environment (variable ↦ location):
node ↦ Loc 8

Store (location ↦ value):
Loc 0 ↦ IntVal 3
Loc 1 ↦ NullVal
Loc 2 ↦ RecVal { data : Loc 0, next : Loc 1 }
Loc 3 ↦ IntVal 2
Loc 4 ↦ RecVal { data : Loc 0, next : Loc 1 } #copy
Loc 5 ↦ RecVal { data : Loc 3, next : Loc 4 }
Loc 6 ↦ IntVal 1
Loc 7 ↦ RecVal { data : Loc 3, next : Loc 4 } #copy
Loc 8 ↦ RecVal { data : Loc 6, next : Loc 7 }
```

The nodes appearing in the `next` chain are not the original nodes allocated by the assign operator, but reconstructed copies allocated by the record fields. Unlike the graph-shaped linked list, identity is not preserved at the node level, since new record nodes are reconstructed during list construction rather than referenced through existing links. Unlike the tree-shaped linked list, previously constructed structure is reused through reconstruction rather than direct embedding, which produces the duplicate records visible in the store.

7.2.3.1 Traversal: alias versus copy

The hybrid list supports two syntactically similar but operationally distinct traversal approaches, demonstrating the difference between environment mutation and store mutation:

```
alias -> node.next.next;
copy  := node.next.next
```

This produces the following changes to the environment and store:

```
Environment (variable ↦ location):
alias ↦ Loc 4
copy  ↦ Loc 9

Store (location ↦ value):
Loc 4 ↦ RecVal { data : Loc 0, next : Loc 1 }
Loc 9 ↦ RecVal { data : Loc 0, next : Loc 1 }
```

Both expressions reach the same logical position in the structure, but they produce different runtime results. `alias` is bound to Loc 4, which is the copy of the first node produced by the `next` field of the second node of the original list structure. Whereas `copy` is bound to Loc 9, which is a fresh location outside the hybrid structure, holding a newly allocated copy of the same value. The environment entry for `alias` points into the list; the environment entry for `copy` points outside it. This distinction has concrete operational consequences. The following programs illustrate what happens when each variable is overwritten with a new node.

Overwriting through `copy` does not affect the original list:

```
copy = @2{data : 9, next : null}
```

```
High-level output:
alias ↦ 3 -> null
copy  ↦ 9 -> null
node  ↦ 1 -> 2 -> 3 -> null
```

Overwriting through `alias` updates the original list:

```
alias = @2{data : 8, next : null}
```

```
High-level output:
alias ↦ 8 -> null
copy  ↦ 3 -> null
node  ↦ 1 -> 2 -> 8 -> null
```

Overwriting through `copy` only affects Loc 9, which is not part of the original list. Overwriting through `alias` affects Loc 4, which is part of the copy chain inside the list, so `node` observes the change. The two mutations look the same syntactically but their effects depend entirely on whether the variable was declared as an alias into the structure or as a copy outside it.

This example demonstrates the phenomenon that causes confusion in languages with implicit aliasing. In Python, a programmer who writes `curr = node.next.next` cannot tell from the syntax whether `curr` aliases an internal node or holds an independent copy. Whether subsequent updates through `curr` affect the original structure is invisible at the syntactic level. In this language, the distinction is explicit: `->` creates an alias, `:=` creates a copy, and `=` always mutates the location it is given.

The language therefore makes the distinction between aliasing and copying explicit through syntax, and forces the programmer to make an intentional choice about which operator to use.

7.2.3.2 Mutation propagation through shared inner locations

Because the hybrid list uses shallow copies, the outer records at each level are distinct, but the inner `data` locations are shared across multiple nodes. This produces a mutation propagation pattern that is more complex than either the graph-shaped or tree-shaped representations.

Consider:

```
alias -> node.next.next;
copy  := node.next;
copy.data  = 9;
alias.data = 8
```

```
Environment (variable ↦ location):
alias ↦ Loc 4
copy ↦ Loc 9
node ↦ Loc 8

Store (location ↦ value):
Loc 0 ↦ IntVal 8
Loc 1 ↦ NullVal
Loc 2 ↦ RecVal { data : Loc 0, next : Loc 1 } # tail
Loc 3 ↦ IntVal 9
Loc 4 ↦ RecVal { data : Loc 0, next : Loc 1 } # copied tail
Loc 5 ↦ RecVal { data : Loc 3, next : Loc 4 } # middle
Loc 6 ↦ IntVal 1
Loc 7 ↦ RecVal { data : Loc 3, next : Loc 4 } # copied middle
Loc 8 ↦ RecVal { data : Loc 6, next : Loc 7 } # head
Loc 9 ↦ RecVal { data : Loc 3, next : Loc 4 }
```

`copy.data = 9` updates Loc 3, which is the location shared by `copy` at Loc 9 and the list nodes at Loc 5 and Loc 7. `alias.data = 8` updates the location referenced by the `data` field of the list node at Loc 4. This location is indirectly reachable through the `next` field of `copy`, which refers to Loc 4. This shows how the mutation

propagates through sharing. The result is that all three variables observe both updates:

```
High-level output:
alias ↦ 8 -> null
copy  ↦ 9 -> 8 -> null
node  ↦ 1 -> 9 -> 8 -> null
```

A somewhat unintuitive consequence of the hybrid construction pattern is that traversal proceeds through reconstructed copies rather than through the originally allocated nodes. Consider the sequence of assignments used to construct a list by repeatedly placing a new node in front of an existing structure. Since the right-hand side of each assignment is evaluated as a value rather than as a location, the previously constructed node is copied into the newly allocated record rather than referenced. As a result, the traversal path reachable from the head node consists of reconstructed nodes created from later allocations. The originally allocated intermediate nodes still exist in the store, but they are no longer part of the primary traversal chain. This differs from graph-shaped linked structures, where traversal follows explicit addresses between independently allocated nodes.

7.2.3.3 Reusing previously constructed lists

The hybrid representation supports constructing larger lists by adding new nodes at the head while reusing an existing list as the remainder of the structure. This allows larger lists to be built incrementally without reconstructing every node in the original list.

```
listA := @2{data : 1, next : null};
listA := @2{data : 2, next : listA};

listB := @2{data : 3, next : listA};
listB := @2{data : 4, next : listB}
```

```
High-level output:
listA ↦ 2 -> 1 -> null
listB ↦ 4 -> 3 -> 2 -> 1 -> null
```

In this example, each assignment allocates only the newly introduced nodes while reusing the previously constructed portion of the list. Because allocation is explicit through the `@n` annotation, it is possible to reason from the program text that constructing `listB` reuses the existing tail of `listA` rather than reconstructing it. The resulting structure resembles persistent list construction found in functional languages, where multiple lists may share a common suffix. However, unlike in functional languages, the shared suffix remains mutable.

Mutating a shared node through one variable therefore becomes visible through all structures that reference the same suffix:

```
listA.next.data = 0
```

```
High-level output:
listA ↦ 2 -> 0 -> null
listB ↦ 4 -> 3 -> 2 -> 0 -> null
```

The examples in this section show that the hybrid representation combines mechanisms commonly associated with different programming paradigms. New nodes are constructed through reconstruction in a manner resembling persistent functional lists, while shared locations remain mutable and therefore exhibit imperative sharing behavior. As a result, the representation supports both structural reuse and observable mutation within the same linked structure.

7.3 Type checker: relaxation and restrictions

The examples in sections 7.3.1-7.3.3 examine different compatibility relations, illustrating the tradeoff between expressiveness and type precision in the language. The examples in section 7.3.4 demonstrate two emergent effects caused by the control-flow restrictions.

7.3.1 Original relaxation for linked list operations

Supporting linked-list traversal and node rewiring through mutation requires pointers to records of different depths to be considered compatible. Under strict structural equality, a pointer to a depth-3 node and a pointer to a depth-2 node have different types. As a result, the write operator rejects updates that attempt to overwrite a pointer to one node with a pointer to another node. Some form of relaxation was therefore necessary to support standard linked-list operations.

The first example demonstrates how the type checker behaves under strict structural equality:

```
# Graph-shaped general list
node := @2{data : true, next : null};
node := @2{data : 2, next : loc node};

curr := loc node;
curr = deref(curr).next
```

```
Type error:

Expected: Addr(Record {data : Int,
                       next : Addr(Record {data : Bool,
                                           next : Null})})

Got: Addr(Record {data : Bool, next : Null})
```

Since one of the `data` field contains a boolean rather than an integer, the structure is treated as a general recursive record rather than a valid graph-shaped linked list. The mutation is rejected because the two nodes have different structural types.

The original relaxation allowed compatibility between pointers into general graph-shaped structures whose shapes differed only in depth. This relaxation was relatively safe for linked lists because every node in a well-formed linked list has `data : Int`, meaning the only field type variation occurs in `next`, which changes predictably with depth. However, the relaxation was unsafe when applied to recursive records in general. Consider the following programs:

```
# Program 1
node := @2{data : 2, next : null};
node := @2{data : 1, next : loc node};

# Program 2
rec := @2{data : 2, next : null};
rec := @2{data : true, next : loc rec}
```

Both programs construct recursive structures, but only program 1 belongs to the graph-list family. Program 2 is a recursive record whose fields happen to form a linked shape. The distinction is important because the original relaxation considered only the outer structure of records rather than recursively validating the entire structure.

The resulting types are:

```
node : Record {data : Int,
               next : Addr(Record {data : Int,
                                   next : Null})}

rec  : Record {data : Bool,
               next : Addr(Record {data : Int,
                                   next : Null})}
```

Consider the following update pattern:

```
p1 := loc rec;
p2 := deref(p1).next;
p2 = p1
```

Under the original relaxation, this mutation was accepted. However, after this mutation, the type environment remains unchanged. The type checker still believes that `p2` points to a record of type:

```
Record {data : Int, next : Null}
```

even though the value now reachable through `p2` has type:

```
Record {data : Bool,
        next : Addr(Record {data : Int,
                            next : Null})}
```

The inconsistency becomes visible when dereferencing `p2`:

```
deref(p2).data = 100
```

Before the mutation, the store contained:

```
Loc 9 ↦ BoolVal True
```

After the mutation, the same location contains:

```
Loc 9 ↦ IntVal 100
```

The type checker accepts this overwrite because it continues to reason using the original type of `p2`. Although `p2` now reaches a different record structure, the type checker still assumes that dereferencing it returns a record whose `data` field has type `Int`. Operationally, however, the mutation overwrites a location that previously contained a boolean with an integer.

This violates the intended invariant that locations preserve their operational structure throughout execution. The underlying problem is that the relaxation causes the type checker to lose track of the structure reachable through a pointer while continuing to reason using the pointer's original type. Linked-list families are less susceptible to this problem because all nodes share the same field types, leaving depth as the only varying component. For arbitrary recursive records, nodes may differ in their field types, allowing pointers to reach store locations whose runtime values do not

conform to their static type.

To address this issue, two helper predicates were introduced, `isGraphList` and `isTreeList`. These predicates recursively inspect an entire structure and verify that every node conforms to the expected linked-list shape before the final relaxation step is applied. Compatibility is therefore restricted to structures that have first been validated as members of a well-defined linked-list family.

7.3.2 Validation of linked-list families

The introduction of recursive validation separates linked structures into two distinct families. Graph-shaped lists use addresses in their `next` field:

```
next : Addr(...)
```

while tree-shaped lists either embed records or copy them:

```
next : Record(...)
```

Consider:

```
rec := @2{data : 1, next : null};
rec := @2{data : 2, next : rec};     # next copies rec
rec := @2{data : 3, next : loc rec}; # next holds address to rec
```

This constructs a mixed linked structure that combines both the tree-shaped and graph-shaped construction pattern. Although such structures are legal values, they are not considered members of either linked-list family. Consequently, the relaxation is not applied.

The recursive predicates ensure that relaxation is restricted to structures that maintain a consistent shape throughout the entire chain. This prevents the type checker from treating tree-shaped and graph-shaped structures as interchangeable.

7.3.3 Symmetric relaxation

After introducing recursive validation through the two helper predicates, a more permissive relaxation was explored to increase the expressiveness of hybrid lists. Pointer-to-pointer compatibility was relaxed for validated lists regardless of depth. Record-to-record compatibility was also relaxed within the same validated list family. This made traversal by mutation and node insertion possible for hybrid lists.

The following example shows a successful insertion of a node in hybrid linked list under the previous symmetric relaxation:

```
node := @2{data : 3, next : null};
node := @2{data : 2, next : node};
node := @2{data : 1, next : node};

# Traverse to tail of the original list
copy := node.next;

# Create a new node to be inserted
newNode := @2{data : 100, next : copy.next};

# Overwrite next with the new node
copy.next = newNode
```

```
High-level output:
node ↦ 1 -> 2 -> 100 -> 3 -> null
```

Note that `newNode`'s `next` field must already contain the intended successor node. Because the relaxation does not permit overwriting `null` with a record, `null` cannot be used as a temporary placeholder and updated later, as is commonly done in graph-shaped lists. The successor must therefore be determined before the insertion is performed.

Although this relaxation supports insertion, it introduces type-safety issues during stepwise traversal of hybrid lists.

Consider the following two traversal patterns on a hybrid list with two nodes:

```
node := @2{data : 2, next : null};
node := @2{data : 1, next : node};

alias -> node;
alias -> alias.next;

curr := node;
curr = curr.next;
```

The type checker assigns the following types:

```
alias : Record {data : Int, next : Null}

curr  : Record {data : Int,
                next : Record {data : Int,
                               next : Null}}

node  : Record {data : Int,
                next : Record {data : Int,
                               next : Null}}
```

The type of `alias` remains precise throughout traversal, whereas the type of `curr` does not. Since the structure reachable through `curr` is no longer tracked precisely, it can be used to change the operational structure of a location.

The overwrite occurs as follows:

```
curr := node;
```

copies the head of the list into Loc 6:

```
Loc 6 ↦ RecVal { data : Loc 3, next : Loc 4 }
```

At the next step, `curr` holds a copy of the tail:

```
curr = curr.next
```

```
Loc 6 ↦ RecVal { data : Loc 0, next : Loc 1 }
```

Although `curr` only contains a copy of the tail node, this copy still shares the location containing the original null terminator. As a result, the following field update becomes possible:

```
curr.next = @2{data : 100, next : null}
```

```
Store (location ↦ value):
Loc 1 ↦ RecVal { data : Loc 7, next : Loc 8 }   # was null
Loc 6 ↦ RecVal { data : Loc 0, next : Loc 1 }   # curr
Loc 7 ↦ IntVal 100
Loc 8 ↦ NullVal
```

Here, the shared reference to the terminating location is followed and the location containing `null` is overwritten with a record value.

This occurs through the same mechanism observed in the previous examples. Once mutation of linked structures is combined with shallow compatibility, there will generally be at least one location whose operational structure may change during mutation. In graph-shaped lists, a simple structural checker cannot simultaneously support linked-list operations and completely prevent null-to-pointer or pointer-to-null conversions. In the hybrid representation, the same limitation appears in a different form, where instead the boundary occurs between record values and null terminators.

Attempting to reach a `null` terminator through traversal and then replace it with a record is not possible, even under the symmetric relaxation:

```
# Method 1: direct alias path
alias -> node.next.next;

# Method 2: direct copy path
copy := node.next.next;

# Method 3: Stepwise alias traversal
alias -> node;
alias -> alias.next;
```

All three traversal patterns preserve the knowledge that the reached location has type `Null`. Consequently, any attempt to overwrite the reached location with a record is rejected. The only situation in which structural null can be overwritten in the hybrid representation is during mutating traversal, where successive updates cause the checker to lose precision about the original location being modified. The final design therefore removes record-level relaxation and restricts the relaxation to validated linked-list pointers.

7.3.4 Emergent effects from control-flow restrictions

Restricting environment-changing operations inside control flow means that all execution paths must preserve compatible assumptions about the structures stored at existing locations.

Consider:

```
x := 1;
if true then x = 2 else x = false
```

Even though the `else` branch would never execute at runtime, the program is rejected:

```
Type error: expected Int, got Bool
```

This occurs because the type checker verifies each branch using the same environment. Since environment-changing operations are forbidden inside control flow, both branches must remain consistent with the same typing assumptions. Branch consistency therefore emerges naturally from the language restrictions.

A similar effect appears for loops:

```
x := 1;

while true do {
    x = false
}
```

```
Type error: expected Int, got Bool
```

The type checker does not need to reason about the number of loop iterations. It is sufficient to verify that a single iteration preserves the existing typing assumptions. If one iteration is type-safe, any finite number of iterations is safe as well.

These two examples illustrate an emergent consequence of the design: environments remain stable within control flow, and existing locations retain a consistent operational structure throughout execution.

7.4 Algorithms and expressiveness

This section presents implementations of common data structure operations and algorithms. The examples also demonstrate how the language exposes allocation, copying, sharing, and mutation behavior that is typically hidden in high-level languages.

7.4.1 Linked-list insertion

Layman et al. identify reference manipulation as a specific obstacle when students implement linked-list operations [5]. The following program demonstrates that pointer creation, pointer traversal, and pointer rewiring are all explicit in the language. Addresses are obtained through `loc`, followed through `deref`, and updated through mutation of the `next` pointer.

```
# Graph-shaped linked list
node := @2{data : 3, next : null};
node := @2{data : 2, next : loc node};
node := @2{data : 1, next : loc node};

# Node to be inserted
new := @2{data : 100, next : null};

# Current pointer
curr := loc node;
done := false;

while (not done) do {
    if deref(curr).data == 2 then {
        new.next = deref(curr).next;
        deref(curr).next = loc new;
        done = true
  } else {
        curr = deref(curr).next
      }
  }
```

```
High-level output:
node ↦ 1 -> 2 -> 100 -> 3 -> null
```

The algorithm traverses the list using a mutable pointer cursor. Once the target node has been found, insertion rewires two pointer fields without allocating new nodes or copying existing structure.

The construction phase and the traversal phase are structurally separated: all allocation happens before the loop, and the loop body contains only mutation and

pointer dereferencing. Because allocation is forbidden inside control flow, the store size after insertion equals the store size before the loop.

7.4.2 Linked-list deletion

Deletion removes a node by changing the links around it rather than modifying the node itself.

```
# Graph-shaped linked list
node := @2{data : 3, next : null};
node := @2{data : 2, next : loc node};
node := @2{data : 1, next : loc node};

curr := loc node;
prev := loc node;
done := false;

# Move current pointer to second node before loop
curr = deref(curr).next;

while (not done) do {
    if deref(curr).data == 2 then {
        # Skip over current node
        deref(prev).next = deref(curr).next;
        done = true
    } else {
        prev = curr;
        curr = deref(curr).next
     }
   }
```

```
High-level output:
node ↦ 1 -> 3 -> null
```

Deletion rewires the `next` field of the previous node to skip the deleted node. The deleted node remains in the store but becomes unreachable from the list, illustrating that removing a node from the logical data structure is not the same as removing it from memory.

The full store after deletion is shown below.

```
Environment (variable ↦ location):
curr ↦ Loc 9
done ↦ Loc 11
node ↦ Loc 8
prev ↦ Loc 10

Store (location ↦ value):
Loc 0 ↦ IntVal 3
Loc 1 ↦ NullVal
Loc 2 ↦ RecVal { data : Loc 0, next : Loc 1 }
Loc 3 ↦ IntVal 2
Loc 4 ↦ AddrVal Loc 2
Loc 5 ↦ RecVal { data : Loc 3, next : Loc 4 }
Loc 6 ↦ IntVal 1
Loc 7 ↦ AddrVal Loc 2
Loc 8 ↦ RecVal { data : Loc 6, next : Loc 7 }
Loc 9 ↦ AddrVal Loc 5
Loc 10 ↦ AddrVal Loc 8
Loc 11 ↦ BoolVal True
```

7.4.3 Bubble sort

Bubble sort is a standard introductory sorting algorithm that repeatedly compares neighboring elements and swaps them until the array is sorted.

```
# Bubble sort
arr := @5[5,2,4,1,3];
n := 5;
i := 0;
j := 0;
temp := 0;

while i < n - 1 do {
    j = 0;
    while j < n - i - 1 do {
        if arr[j] > arr[j+1] then {
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp
        } else {
            skip
        };
        j = j + 1
    };
    i = i + 1
}
```

```
User-level (variable ↦ value):
arr ↦ [1 2 3 4 5]
i ↦ 4
j ↦ 1
n ↦ 5
temp ↦ 2
```

The array is allocated before execution and the loop only mutates existing positions in the array. The extra location required for swapping values is introduced through `temp`, making all storage used by the algorithm visible before execution begins.

Since the language does not have scope, `j` has to be reset to 0 manually at the start of each outer loop iteration, so that each pass begins scanning the array from the first element.

7.4.4 Shifting elements in an array

Unlike linked-list insertion, inserting into the front of an array requires moving existing elements. The following example makes that cost visible through the sequence of updates performed by the program.

```
# Array with extra capacity (0's are placeholders)
arr := @6[2,3,4,0,0,0];

size := 3;     # logical array size

# Initialise cursor at last valid element
i := size - 1;

while i >= 0 do {
    arr[i+1] = arr[i];
    i = i - 1
};

# Insert new value at front
arr[0] = 1;

size = size + 1
```

```
User-level: variable ↦ value
arr ↦ [1 2 3 4 0 0]
i ↦ -1
size ↦ 4
```

Inserting at the front requires shifting every existing element one position to the

right before writing the new value. The traversal must go right to left because a left-to-right traversal would overwrite values before they are copied. This is observable from the program: each iteration reads a position and immediately overwrites the next position. The cost is linear in the number of existing elements. By contrast, inserting at the front of a linked list requires only a constant number of pointer updates.

7.4.5 Partial mutation

The final example demonstrates partial mutation, where different structures share only parts of their underlying representation, so that a single update propagates through some structures but not others.

```
# Values to be shared
v1 := 1;
v2 := 2;
v3 := 3;

# Base array
arr := @3[loc v1, loc v2, loc v3];

winA := @2[arr[0], arr[1]];
winB := @2[arr[1], arr[2]];

# Mutate shared element
deref(arr[1]) = 9
```

```
Final state (variable ↦ location ↦ value):
arr  ↦ Loc 6 ↦ [Loc 0 Loc 1 Loc 2]
v1   ↦ Loc 0 ↦ 1
v2   ↦ Loc 1 ↦ 9
v3   ↦ Loc 2 ↦ 3
winA ↦ Loc 9 ↦  [Loc 0 Loc 1]
winB ↦ Loc 12 ↦ [Loc 1 Loc 2]
```

The array `arr` is constructed using `loc` expressions, meaning its elements store pointers to the locations of `v1`, `v2` and `v3` rather than copies of their values. When `winA` and `winB` are constructed, these pointer values are copied into the new arrays. This does not duplicate the underlying values, but instead preserves sharing of the referenced locations across all three structures. Mutating through `deref(arr[1])` updates that shared location, making the change visible through all three arrays simultaneously. The explicit use of `loc` makes these pointer-sharing relationships visible in the program text, allowing the programmer to reason about whether mutations will propagate across structures.

8

Discussion

The design presented in this thesis emerged from the goal of making memory effects underlying mutable data structures explicit. The motivation for this came from prior studies discussed in the background section, which identified hidden semantics and overloaded operators as factors that complicate reasoning about mutable structures. The decision to separate variable introduction, aliasing and mutation occurred incrementally during the design process.

In languages where operators are overloaded, the syntax directs attention toward surface structure rather than the memory effects that actually determine program behavior. The present work attempts to address this at the level of language design by making hidden semantics visible in the program text. Many of the resulting language properties can be traced to the decision to make allocation of expressions uniform, which in turn was a consequence of the design principle that semantics should not be hidden. In particular, expression allocation was made uniform in order to ensure that the operator responsible for copying values would not implicitly create sharing depending on context.

Sharing within mutable data structures has been consistently identified as one of the more difficult aspects of reasoning about imperative programs. In order to represent sharing while simultaneously exposing the behavior explicitly, two things were needed: compound values had to contain locations and allocation had to be unconditional. Sharing is unavoidable when compound values containing locations are copied, unless copying is made deep by default. However, making copying deep by default would remove sharing rather than expose it.

Making allocation of expressions explicit in the syntax was a design choice motivated by the fact that the effects of shallow copying are difficult to reason about if you do not know that compound values may contain locations rather than store values directly. The choice to expose this syntactically was a compromise between operational simplicity and making the effect of shallow sharing visible without requiring full knowledge of the underlying semantics.

Since each operator has a fixed and context-independent meaning, reasoning about the effect of an operation depends primarily on understanding the structures involved

rather than on determining which behavior an operator might trigger depending on the values at runtime. This connects to the discussion on overloaded operators, assignment semantics, and notional machines in the background section, especially to Fincher et al.'s argument that effective learning requires knowing which features of an execution model to attend to [8]. While not intended as a necessary part of the language, the store output serves an analogous function during the evaluation of the design: it makes the resulting memory structure observable, allowing the consequences of semantic and syntactic choices to be examined concretely.

The result is a language in which allocation behavior is both deterministic and visible, making sharing relationships inferable from program structure. This does not eliminate the need to reason about sharing. As demonstrated by the hybrid linked list, complex sharing patterns can still emerge and may be difficult to reason about. However, those patterns arise from explicit construction and allocation behavior rather than from hidden effects.

8.1 Memory visibility and control-flow restrictions

The language makes several aspects of memory behavior visible. Since allocation is represented explicitly in the syntax, the number of newly created locations can be determined from the program text. This allows programmers to reason about memory growth without having to infer allocation behavior from language-specific assignment semantics. Because the core operators have distinct meanings, programmers do not need to determine whether an operation creates an alias or updates an existing structure. The separation of allocation, aliasing and mutation, together with explicit pointers, exposes how programs construct and manipulate memory. As a result, reasoning about the source code reveals whether a computation creates new structure, reuses existing structure, or introduces sharing relationships.

Although the language primarily exposes memory behavior, some costs relevant to time and space complexity are also reflected in the syntax. In particular, the distinction between shallow and deep copying becomes visible. A shallow copy allocates only a new outer location while reusing existing inner locations. By contrast, a deep copy requires the allocation of new inner locations and the explicit reconstruction of the copied structure. The amount of work performed therefore grows with the size of the copied structure, whether the copy is expressed explicitly or through iteration. Deep and shallow copying underlie many phenomena encountered in practice, including object copying, shared mutable structures and structural sharing. The language makes many of the memory effects that contribute to complexity visible in the program text, although it does not eliminate the need for asymptotic analysis.

The control-flow restrictions strengthen the separation between construction and processing. Restricting allocation and alias creation inside control flow limits certain forms of dynamic program construction, such as constructing linked structures using loops. However, the restriction does not appear to significantly reduce expressiveness

within the intended domain, which is restricted to basic mutable data structures. Within this relatively narrow setting, the language still supports the operations that are necessary: traversing structures, performing updates, and inspecting sharing relationships.

Beyond type safety, the restrictions may also improve the reasoning properties of programs. Allowing unrestricted alias and variable rebinding inside loops can make programs harder to reason about. The main reason for this is that the assign and alias operators are more permissive than the write operator, since they can freely rebind names to arbitrary locations without preserving structural constraints. Keeping track of this within loops becomes especially difficult, since it means that the same name may refer to different locations and associated types, potentially altering relationships between variables and locations on each iteration. Mutation behaves differently in this respect. It operates on an already existing location without changing which location a name refers to, so the type assumptions established before the loop remain stable across iterations.

8.2 Reasoning about reconstruction and sharing

The three different linked list constructions demonstrate that memory topology emerges from construction style rather than from specialized language features or context-dependent evaluation rules. Both the graph-shaped list produced by referential construction and the tree-shaped list produced by embedding have relatively transparent correspondences between program text, memory topology and traversal style. The hybrid construction breaks this correspondence. Despite operators being distinct and uniform, and sharing relationships being inferable from the program text as well as visible in the store, the resulting structure is difficult to reason about. The reason for this is that the hybrid linked list combines value construction with shallow sharing, which means that the structure of the program and actual memory representation diverge.

What makes the hybrid list particularly interesting is that this complexity is a direct consequence of making effects explicit. The construction pattern emerges because allocation of compound values is uniform. The reference prototype, which shares the core design and the property that compound values contain locations while differing only in allocation rules, does not produce the same memory topology, which confirms that the hybrid list representation is a direct consequence of making allocation uniform.

The hybrid list appears as a normal linked list in the program text, and supports syntactically simple traversal patterns. However, the memory representation contains both reconstructed copies and shared inner locations simultaneously. When traversing from the head, execution moves through original nodes and into copies created by evaluating `next` fields, without any obvious indication in the high-level structure that this transition has occurred. Johnson et al. identify this kind of disconnect between program structure and underlying memory behavior as a source of persistent

confusion in high-level languages [3], while Sibia et al. characterize this as a structural abstraction problem, where the difficulty lies not in individual operations but in connecting their cumulative effects to the high-level structure they implement [10]. The hybrid demonstrates that this difficulty does not disappear when the operations themselves are explicit. Even with explicit allocation, explicit aliasing, and visible store effects, a construction that combines reconstruction with sharing remains difficult to reason about.

Rather than arising from hidden operations, the difficulty arises from the interaction between visible operations whose effects propagate through the reconstructed structure. The hybrid representation therefore suggests that some of the difficulties associated with reasoning about mutable data structures arise from the interaction between sharing and reconstruction rather than solely from hidden language semantics.

8.3 Type checker limitations

Linked-list operations require some degree of type checker relaxation in this system. Under strict structural equality, ordinary traversal and update operations become difficult to express because nodes at different depths have different types. However, increasing flexibility also makes it more difficult for the type checker to maintain precise information about the structure reached through mutable references.

The original relaxation demonstrated that excessive flexibility can allow the checker to reason using outdated structural information after mutation. In particular, allowing compatibility between pointers to records that differed only in structural depth meant that mutations could invalidate structural assumptions without the type checker detecting it. The null-to-record conversion in the hybrid list under the previous symmetric relaxation illustrated a related limitation. During mutating traversal, the type checker may lose precision about the structure currently being reached. In the hybrid representation, shallow sharing allows this loss of precision to affect multiple structures that reach the same location.

The most challenging cases for the type checker arise from mutable structures that contain references to other locations. This applies whether those references are introduced through explicit pointers or through the inner location sharing that occurs in compound values.

This suggests that the primary limitation is not pointers themselves, but maintaining precise structural information during mutating traversal of recursive structures. Supporting linked-list operations while preserving precise structural information therefore becomes difficult within a simple structural type system. A type system with store typing or recursive types would likely be required to handle these cases precisely.

8.4 Summary of research questions

RQ1 asked how separating allocation, aliasing, and mutation into distinct syntactic operators can support reasoning about mutable data structures and memory effects. The behavior of mutable data structures depends on allocation, aliasing, mutation, and sharing relationships. By making these effects visible or inferable in the program text, the language exposes information that is relevant to understanding and predicting mutable data structure behavior. Knowing whether an operation creates an alias or mutates an existing location is relevant to reasoning about whether a line of code will introduce a new relationship between two variables or if it will update the value of an existing variable. Similarly, exposing where allocation happens in a program supports reasoning about how sharing relationships are created, maintained, and broken as a program executes. The language makes a visible distinction between operations that allocate new structures (through the assign operator and explicit expression allocation), introduce new names for existing locations (through aliases) or create references to existing locations (through pointers). Distinguishing between these semantically distinct operations provides information that is relevant to reasoning about the correspondence between programs and memory effects.

RQ2 asked how different syntactic construction patterns induce different memory topologies and sharing behavior. The linked-list examples demonstrate that memory topology emerges from construction style. Different syntactic construction patterns induce different memory topologies because allocation occurs at different points during evaluation. Referential construction allocates nodes independently and connects them through pointers, producing graph-shaped structures. Embedding allocates locations as part of nested expression evaluation, producing tree-shaped structures. Hybrid constructions combine reconstruction and sharing, producing more complex topologies that contain both copied structure and shared locations. Memory topology and sharing behavior therefore emerge from construction style and allocation behavior.

RQ3 asked what limitations emerge when using a simple structural type system together with mutable pointer-based structures. The results show that the primary limitation is a loss of type precision during mutating traversal of linked structures. Repeated mutation of traversal variables can cause structural assumptions to become outdated, which causes the type checker to reason using types that no longer accurately reflect the structure currently being reached.

9

Conclusion

The language presented in this thesis makes memory effects that are often implicit in high-level programming languages explicit. Separating variable introduction, aliasing, and mutation into distinct operators makes it possible to distinguish between different kinds of memory effects in the program text and shifts reasoning from interpreting overloaded operators to understanding the effects they produce. The design also makes allocation behavior visible and deterministic, allowing sharing relationships to be inferred from program structure.

The results further demonstrate that different construction patterns produce different memory topologies and sharing behavior despite the underlying evaluation model remaining unchanged. In particular, the graph-shaped, tree-shaped, and hybrid linked-list representations show how relatively small syntactic differences can lead to very different structural and operational properties.

Finally, the type checker results show a tradeoff between type precision and expressiveness when supporting mutation in linked structures. A simple structural type checker can either be strict and reject ordinary linked-list operations, or be more permissive and lose precision about mutable shared structure. The hybrid list demonstrates that this limitation is not specific to explicit pointers, but arises whenever mutation and linked structures with shared references interact.

Future work could extend the type checker with store typing, recursive types and nullable pointer types to handle the cases in which the current type checker loses precision. Alternative design decisions could be explored such as allowing pointers to take an lvalue instead of an identifier, which would increase expressiveness but make the language more low-level. Restricting variable rebinding to preserve the existing type could also be explored. This would permit alias-based traversal inside loops and remove some of the current control-flow restrictions. More broadly, the language could be extended with additional primitives to cover a wider range of data structures and memory behaviors.

Bibliography

[1] Dimitri Racordon and Didier Buchs. *Explicit and Controllable Assignment Semantics*. 2019. arXiv: 1907.11317 [cs.PL]. URL: https://arxiv.org/abs/1907.11317.

[2] Dimitri Racordon et al. "Implementation Strategies for Mutable Value Semantics". In: *J. Object Technol.* 21 (2022), 2:1–11. URL: https://api.semanticscholar.org/CorpusID:248335097.

[3] Fionnuala Johnson, Stephen McQuistin, and John O'Donnell. "Analysis of Student Misconceptions using Python as an Introductory Programming Language". In: *Proceedings of the 4th Conference on Computing Education Practice*. CEP '20. Durham, United Kingdom: Association for Computing Machinery, 2020. ISBN: 9781450377294. DOI: 10.1145/3372356.3372360. URL: https://doi.org/10.1145/3372356.3372360.

[4] "Hints on programming-language design". In: *Essays in Computing Science*. USA: Prentice-Hall, Inc., 1989. ISBN: 0132840278.

[5] Lucas Layman, Toni Pence, and Sridhar Narayan. "Topic Difficulty in a Python Data Structures Course: Student Perceptions and Performance". In: May 2025, pp. 163–170. DOI: 10.1145/3696673.3723056.

[6] Daniel Zingaro et al. "Identifying Student Difficulties with Basic Data Structures". In: Aug. 2018, pp. 169–177. DOI: 10.1145/3230977.3231005.

[7] Syeda Mazumder and Manuel Pérez-Quiñones. "Incoming CS1 Students' Misconceptions on Arrays". In: Oct. 2023, pp. 1–9. DOI: 10.1109/FIE58773.2023.10342927.

[8] Sally Fincher et al. "Notional Machines in Computing Education: The Education of Attention". In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '20. Trondheim, Norway: Association for Computing Machinery, 2020, pp. 21–50. ISBN: 9781450382939. DOI: 10.1145/3437800.3439202. URL: https://doi.org/10.1145/3437800.3439202.

[9] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. "Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '17. Seattle, Washington, USA: Association for

Computing Machinery, 2017, pp. 213–218. ISBN: 9781450346986. DOI: 10.1145/3017680.3017777. URL: https://doi.org/10.1145/3017680.3017777.

[10] Naaz Sibia et al. "From State to Structure: Towards Abstraction Support in CS2". In: *Proceedings of the 25th Koli Calling International Conference on Computing Education Research*. Koli Calling '25. Association for Computing Machinery, 2025. ISBN: 9798400715990. DOI: 10.1145/3769994.3769998. URL: https://doi.org/10.1145/3769994.3769998.

[11] Liat Nakar, Mor Friebroon-Yesharim, and Michal Armoni. "From Modelling to Assessing Algorithmic Abstraction – the Missing Dimension". In: Feb. 2024, pp. 1–12. DOI: 10.1145/3631802.3631815.

[12] Duane J. Jarc. "Data Structures: a Unified View". In: *ACM SIGCSE Bull.* 26 (1994), pp. 2–4. URL: https://api.semanticscholar.org/CorpusID:15179933.

[13] Ruslan Batdalov and Oksana Nikiforova. "Patterns for Assignment and Passing Objects Between Contexts in Programming Languages". In: *Proceedings of the 26th European Conference on Pattern Languages of Programs*. EuroPLoP '21. Graz, Austria: Association for Computing Machinery, 2022. ISBN: 9781450389976. DOI: 10.1145/3489449.3489975. URL: https://doi.org/10.1145/3489449.3489975.

[14] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *Proceedings of the 15th International Workshop on Computer Science Logic*. CSL '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 1–19. ISBN: 3540425543.

[15] James Noble et al. "Designing Grace: Can an introductory programming language support the teaching of software engineering?" In: *2013 26th International Conference on Software Engineering Education and Training (CSEET)*. 2013, pp. 219–228. DOI: 10.1109/CSEET.2013.6595253.

[16] Diomidis Spinellis. "Notable Design Patterns for Domain-specific Languages". In: *Journal of Systems and Software* 56 (Feb. 2001), pp. 91–99. DOI: 10.1016/S0164-1212(00)00089-3.

[17] Marjan Mernik Tomaž Kosar and Jeffrey C Carver. "Program Comprehension of Domain-specific and General-purpose Languages: Comparison Using a Family of Experiments". In: *Empirical software engineering* (2012).

[18] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: a minimal core calculus for Java and GJ". In: *Conference on Object-Oriented Programming Systems, Languages, and Applications*. 1999. URL: https://api.semanticscholar.org/CorpusID:52798834.

[19] Benjamin C. Pierce et al. *Logical Foundations*. Software Foundations Series, Volume 1. Version 5.5. http://www.cis.upenn.edu/ bcpierce/sf. Electronic textbook, May 2018.

[20] Glynn Winskel. *The Formal Semantics of Programming languages: an introduction*. Cambridge, MA, USA: MIT Press, 1993. ISBN: 0262231697.

[21] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091.