software

MDPI

Article

A Brief Overview of the Pawns Programming Language

Lee Naish

Computingand Information Systems, The University of Melbourne, Parkville 3052, Australia; dr.lee.naish@gmail.com

Abstract: This paper describes the Pawns programming language, currently under development, which uses several novel features to combine the functional and imperative programming paradigms. It supports pure functional programming (including algebraic data types, higher-order programming and parametric polymorphism), where the representation of values need not be considered. It also supports lower-level C-like imperative programming with pointers and the destructive update of all fields of the structs used to represent the algebraic data types. All destructive update of variables is made obvious in Pawns code, via annotations on statements and in type signatures. Type signatures must also declare sharing between any arguments and result that may be updated. For example, if two arguments of a function are trees that share a subtree and the subtree is updated within the function, both variables must be annotated at that point in the code, and the sharing and update of both arguments must be declared in the type signature of the function. The compiler performs extensive sharing analysis to check that the declarations and annotations are correct. This analysis allows destructive update to be encapsulated: a function with no update annotations in its type signature is guaranteed to behave as a pure function, even though the value returned may have been constructed using destructive update within the function. Additionally, the sharing analysis helps support a constrained form of global variables that also allows destructive update to be encapsulated and safe update of variables with polymorphic types to be performed.

Keywords: programming language; functional programming; effects; destructive update; mutability; sharing; aliasing

Citation: Naish, L. A Brief Overview of the Pawns Programming Language. *Software* **2024**, *3*, 473–497. https://doi.org/10.3390/software3040023

Academic Editor: Vincenzo Gervasi

Received: 15 September 2024
Revised: 9 November 2024
Accepted: 14 November 2024
Published: 19 November 2024

Copyright: © 2024 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

This paper briefly describes the main features of the Pawns programming language (https://lee-naish.github.io/src/pawns/ (accessed on 19 November 2024)). It is currently being developed solely by the author and is intended as a "proof of concept" for various new language features rather than a fully featured language ready for deployment. The aim of this paper is to give an overview of the main new ideas (Ref. [1] does the same but includes significantly more detail and discussion of the language design). Here, we attempt to avoid becoming too bogged down with such issues, at least for the earlier parts of the paper. We assume the reader is familiar with Haskell [2] and C [3]. Pawns supports pure functional programming with strict evaluation, algebraic data types (ADTs), parametric polymorphism and higher-order programming. It also supports "impure" code, such as using state (including IO) and the destructive update of all compound data types via pointers (references or "refs" for short) but all such code is highlighted by "!" annotations. A call to a function that relies on state must be prefixed by "!"; the details of the state(s) are declared in the type signature. Additionally, variables that are updated must be prefixed with "!". A function call with no "!" is guaranteed to behave as a pure function, though Pawns allows impurity to be encapsulated (and checked by the compiler), so the function may be implemented using impure features. The representations of different variables can be shared, so updating one variable may also update other variables and the Pawns compiler checks that all relevant variables are annotated with "!" at that point in the source code: Pawns is an acronym for "Pointer assignment without nasty surprises" and

its most important (and complex) innovation is the way update of shared data structures is supported, and how pure and impure code can be mixed. Impure programming in Pawns can be like programming in C, with the destructive update of fields of structs representing ADT values and performance equal to or better than portable C. However, there are no unsafe operations (such as dereferencing possibly NULL pointers, casts, accessing fields of unions, etc.) and all interactions/dependencies due to sharing must be documented in annotations/declarations.

The rest of this paper is structured as follows. Section 2 gives a little more detail on the motivation for developing yet another programming language in this style and clarifies our aim. Sections 3–10 present the main features of Pawns that are of interest. Section 3 gives a simple example of pure functional programming supported in Pawns. Section 4 describes data representation, how pointers (refs) can be created and how they can be used for destructive update, with "!" annotations on statements. Section 5 gives two examples of code using destructive update in Pawns, mentioning sharing of data structures but deferring the details of how sharing is handled. Section 6 discusses the distinction between data structures that can be simply viewed as abstract values (typical in pure code) and those for which sharing must be understood (a necessity when destructive update is used). Section 7 discusses how sharing and destructive update information is incorporated into Pawns type signatures and the kind of sharing analysis performed by the compiler. Section 8 presents how IO and other forms of "state" similar to global variables can be used in Pawns. Section 9 discusses a Pawns feature that allows the renaming of functions so different type signatures can be given, overcoming some of the limitations of polymorphism, particularly for impure Pawns code. Section 10 briefly discusses some of the additional complications surrounding safety in Pawns, including the foreign function interface. Section 11 discusses further work. Section 12 compares Pawns to some related programming languages. Section 13 summarises the main contributions of Pawns. Section 14 discusses threats to validity of these contributions. Section 15 concludes.

2. Motivation and Aim

The design of pure functional programming languages helps us follow some of the main lessons learned from software engineering: large problems are best decomposed into relatively small problems and solved with small code units (functions) that are largely independent of each other. Reducing dependencies between components and making all dependencies obvious reduces system complexity, a difficult problem of software engineering. Pure functions simply take values as arguments and return a value as a result, with no other hidden dependencies or effects, and variables denote values rather than memory locations whose contents may change. Program execution is simply evaluating expressions by calling functions, and the type of a function gives a summary of what calling the function does. The type system, including algebraic data types, parametric polymorphism and higher-order functions, provides a very expressive domain for encoding values, allows the compile time detection of many errors, and supports code re-use and abstraction. However, for structured data there are disadvantages of the pure functional approach. Instead of being able to destructively update part of a data structure with a new value replacing an old value (such as assigning a new value to an array element, a field of a struct or something more complex such as a node in a tree), a new version of the whole data structure has to be constructed. All parts that contain the updated value must be copied (conceptually at least); other parts can be shared between the old and new data structures because the value of these parts has not changed. This can be more cumbersome to code and difficult to implement efficiently. Incorporating IO into the functional programming paradigm is also challenging.

In imperative programming languages, variables typically denote values *and* memory locations that contain the values (known as r-values and l-values, respectively). Assigning values to memory locations is one of the fundamental operations, so the destructive update of data structures is easy, natural and efficient. IO is also natural. However, having a notion

of a memory store can greatly increase subtle dependencies between different code units. This is why "global variables" are best used sparingly. Also, different variables can use the same memory locations to store (parts of) their values, particularly when pointers are used. Two pointer variables may be *aliased* (that is, they both point to the same memory location) and two variables whose values are compound data structures may *share* parts of their representations (for example, pointers within the data structures may be aliased). Assigning a new value to a memory location associated with one variable can change the value of other variables as well. Thus, the sharing of mutable data structures introduces additional subtle dependencies between code units. Programmers must be aware of the memory layout of data structures (including any sharing), not simply the value represented. There can also be classes of errors that do not exist in pure functional languages such as dereferencing NULL pointers, buffer over-runs, reading from uninitialised memory and errors associated with dynamic memory allocation.

Although the combination of pointers, sharing and destructive update can potentially result in nightmarish complexity, it is also *very* useful. There are important algorithms that rely on the destructive update of shared values. For example, graph reduction [4] is a fundamental implementation technique for functional programs that uses a directed acyclic graph to represent an expression. There can be multiple pointers to a node representing an expression, and the node must be destructively updated to avoid the evaluation of the expression multiple times. Similarly, in the logic programming language Prolog, variables can be aliased and later instantiated, so implementations update shared structures [5,6]. Unification and finding the value of Prolog variables can be seen as an instance of the ubiquitous union-find (or disjoint-set) problem, for which the best algorithms update shared structures [7]. There are also situations where the destructive update of shared structures can make code simpler. For example, consider a program that uses a representation of an environment containing several objects with various attributes such as object identity, object type, position and other state information that may vary over time. It may be convenient to access them via an array (given the object identity), a priority queue (for the next object to consider in a discrete event simulation), a quad tree (for object positions), et cetera . Without a shared representation of objects, any change to the state of an object requires *all* such indexing structures to be modifed. A longer discussion of these issues is contained in [8].

If a function has some "effect", such as updating a data structure, it is not simply a mathematical function mapping its argument values to a result—it is "impure". Eliminating impurity completely can make it difficult to implement some algorithms efficiently, leading to frustration for programmers and cries such as "I love purity, but it's killing me" (https://www.haskell.org/pipermail/haskell-cafe/2008-February/039339.html (accessed on 13 November 2024)).

Various languages have been designed to support both pure and impure code (we defer most of our discussion on this to Section 12) but it is difficult to support impurity without it "leaking out" into surrounding code. First, if a function is impure, any function that directly or indirectly calls it is also potentially impure. It is desirable to be able to encapsulate the impurity so that a function *behaves* in a pure way for any function calling it, even though it may use effects internally. Second, in several languages, mutability is a property of a data structure or a data type, and any function that uses a mutable data structure is potentially impure. Adding destructive update to code may require changing the types so the code is less efficient and potentially copying data structures.

The aim of Pawns is to develop a language with the advantages of pure functional programming that also allows flexible low-level programming involving pointers and the update of shared data structures. Even if a Pawns function call does not behave as a pure function, simply mapping argument values to the result, any additional dependencies and effects are made obvious in the source code; this is called "interface integrity" in [9]. The type signature of a function can be used to determine whether it behaves as a pure function, but a pure function may encapsulate impurity in its implementation. Data types

do not need to be changed in order to support mutability. There is flexible support for the update of all compound data types, with no unsafe operations such as NULL pointer dereferencing (the one exception is the foreign language interface). A data structure can be created using destructive update then passed to other code without the risk that the other code will update it further, encapsulating the impurity. There is also a constrained form of global variables that allows encapsulated destructive update and support for IO. Even if Pawns is never fully developed into a practical language, we hope some of its novel features will be influential.

3. Pure Functional Programming Example—BST Creation

Consider the task of converting a list of integers into a binary search tree. Pawns supports typical pure functional programming solutions such as Figure 1, presented using Haskell-like syntax (Pawns currently only supports a temporary syntax, to avoid decisions on syntax and the need to write a parser). It uses the algebraic data type `BST`. Values of this type are either the data constructor `Empty`, representing an empty tree, or the data constructor `Node` with three arguments of type `BST`, `Int` and `BST`, respectively. The code includes the polymorphic `List` type definition (pre-defined in Pawns), which is used to defined type `Ints` (lists of integers). The code also shows how the standard library function `foldl` can be defined. It is polymorphic (allowing lists of any type) and higher-order (the first argument is a function). An advantage of this style of programming is that it is not necessary to understand how values are represented in order to write and reason about the code. However, `bst_insert_pure` builds a new node at each level of the tree visited. This is less efficient and arguably more complicated than simply using destructive update when a leaf is reached (see Section 5).

```
data BST = Empty | Node BST Int BST -- binary search tree of integers
data List t = Nil | Cons t (List t) -- polymorphic Lists (built in)
type Ints = List Int                -- list of integers

-- convert list of integers to BST (pure code)
list_bst_pure:: Ints -> BST
list_bst_pure xs =
    foldl bst_insert_pure Empty xs

-- insert integer into a BST to give new BST
-- (pure; re-builds a path from root to a leaf)
bst_insert_pure:: BST -> Int -> BST
bst_insert_pure t0 x =
    case t0 of
    Empty ->
        Node Empty x Empty
    (Node l n r) ->
        if x <= n then
            Node (bst_insert_pure l x) n r
        else
            Node l n (bst_insert_pure r x)

-- standard library foldl for lists
foldl:: (b -> a -> b) -> b -> List a -> b
foldl f y xs =
    case xs of
    Nil ->
        y
    (Cons x xs1) ->
        foldl f (f y x) xs1
```

Figure 1. BST creation using pure code.

4. Representation and Destructive Update of Values

The key thing to note about data representation and update in Pawns is that *arguments of data constructors are stored in main memory* and these are the only things that can be updated. A data constructor with one or more arguments is like a pointer (or a pointer plus a "tag") to a block of words containing the argument(s). One with no arguments is represented as a small integer. The list `(Cons 2 Nil)` is represented as a pointer to two memory cells, containing 2 and 0, respectively—identical to C. Similarly, a BST is represented identically to C code using pointers to structs with three fields. In Section 5, we show code that manipulates a `Cord` type that is either a `Branch` data constructor containing two cords or a `Leaf` data constructor containing a list. The Pawns representation of cords uses a one-bit tag on the pointers to distinguish the two different data constructors because both have arguments so they must both be pointers. This is more efficient than portable C code (which requires a union of structs); see [10] for details. There is also a special `Ref` type that has a single data constructor with one argument.

Figure 2 gives an example of the memory layout. Boxes represent main memory words, data constructors followed by a pointer represent tagged pointers (for `Cons` and `Ref`, the tag is empty) and `Nil` is represented by zero. The variable `xc` is a cord of the form `Branch (Branch (Leaf l1) (Leaf l2)) (Leaf l34)`, where `l1` is the list `[1]`, `l2` is the list `[2]` and `l34` is the list `[3,4]`. The variable `xsp` is a reference to a word containing `Nil` (see below).

Pawns allows the kind of programming we can carry out in C with pointers to structs and assignment to fields of structs. There is also additional flexibility because an ADT can have any number of data constructors with arguments (which is like having a pointer to any number of different struct types) and any number of data constructors with no arguments (like have any number of different NULL values). Furthermore, there are no unsafe operations such as dereferencing NULL pointers, casts, unions, et cetera.

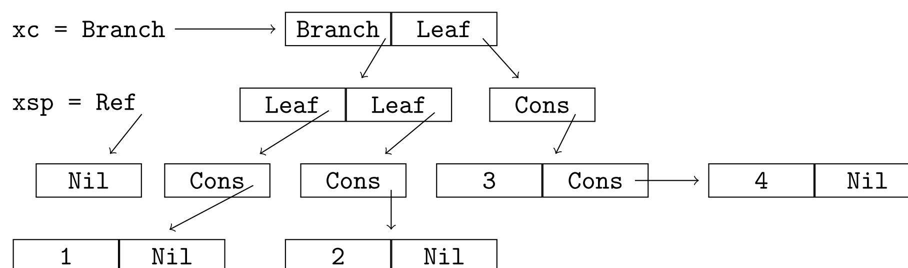

Figure 2. Data layout showing tagged pointers.

Pawns variables are not names for memory locations that can be updated—it is not possible to assign to an existing variable or obtain a "pointer to a variable" as can be performed in C. That is why the variables `xc` and `xsp` in Figure 2 are not in boxes. However, the representation of the value of a variable may have mutable components. For example, the variable `xc` will always be a `Branch` pointer to the same two memory cells but the content of these cells and/or those they point to can potentially be updated, changing the overall value of `xc`. All update is carried out via the (polymorphic) `Ref` type (similar to `STRef` in Haskell and `ref` in ML [11]). A value of type `Ref t` is a pointer to a memory cell containing a value of type `t`. You can think of the memory cell as the argument of the data constructor for the `Ref` type; thus, it can be updated. Though Figure 2 shows the data constructor as `Ref`, Pawns code never uses an explicit data constructor for refs but instead simply uses a dereference operator, "`*`", like C. If `x` is a Pawns expression of type `Ref t` then `*x` is the value of type `t` that `x` points to. There are no `NULL` refs.

The simplest way to create a ref is by using a let binding with `*` prefixing the let-bound variable. The "let" and "in" keywords of Haskell are not required in Pawns and ";" is used for sequencing (like the "do" notation in Haskell); thus, `x = 42; *xp = 42` creates two variables, the first of which equals 42 and the second points to a newly allocated

memory cell containing 42 (similar to the Haskell monadic code `xp <- newSTRef 42`, or an ML let expression with `xp = ref 42`). Destructive update is performed by dereferencing a pointer on the left of the "`:=`" (assignment) operator. A "!" prefix must be included for all affected variables (more precisely, all affected live variables; those which are never used again can generally be ignored). Typically, there will be a pointer variable on the left (so `*xp := ...` is written `*!xp := ...`) but there may also be other variables that share its representation; these can be annotated with "!" at the right of the statement. Figure 3 has a simple example.

```
x = 42;          -- let binding of x to 42
*xp = x;         -- xp points to a new memory cell containing 42
yp = xp;         -- yp points to the same memory cell (aliases xp)
y = *yp;         -- y is the contents of the memory cell (42)
*!xp := 43 !yp; -- update what xp points to (also affects yp!)
z = *yp          -- z is the contents of the memory cell (43)
```

Figure 3. Destructive update via a ref.

Without the "`!y`" annotation, we would have `y = *yp` and `z = *yp` with no intervening occurrence of `yp` in the code; yet, `y` and `z` end up with different values. This is a small example of the potentially confusing "surprises" encountered in languages that support code for destructive update with pointer aliasing and shared data structures. Both the sharing between `xp` and `yp` and the update of `xp` could have been performed by function calls, further increasing the subtlety. Pawns supports such code but insists the programmer documents sharing and effects, in a way that is checked by the compiler.

Just as prefixing a variable with `*` in a let binding creates a pointer variable, the same can be performed with pattern bindings. These "reference patterns" are an important innovation of Pawns. For example, the code for `bst_insert_pure` could be rewritten as in Figure 4. Instead of the pattern matching with a `Node` creating variables of type BST and Int, it creates variables of type `Ref BST` and `Ref Int`, which are pointers to the arguments of the `Node` data constructor. Refs are created but no extra memory cells are allocated and no monads or changes to the `BST` type are required. There is no equivalent in languages such as Haskell and ML, but Disciple [8] (now called Discus) allows the creation of references to arguments by using named fields rather than pattern matching. The subsequent code simply dereferences the pointers to obtain the same values as before and the code is pure—refs/pointers themselves do not introduce impurity. However, such pointers could potentially be used to destructively update the `Node` arguments (which is impure).

```
bst_insert_pure_p t0 x =
    case t0 of
    Empty ->
        Node Empty x Empty
    (Node *lp *np *rp) ->    -- creates refs/pointers to Node arguments
        if x <= *np then
            Node (bst_insert_pure_p *lp x) *np *rp
        else
            Node *lp *np (bst_insert_pure_p *rp x)
```

Figure 4. BST insertion using pure code with pointers.

5. Destructive Update Examples

We now give two short examples of using destructive update in Pawns. The first is an alternative way to construct a BST and the second is an example where the sharing of data structures is more complex. Building a BST from a list of integers can be performed very efficiently by first allocating a memory cell containing an empty BST then repeatedly

traversing down the tree and destructively inserting the next integer as a new leaf—see Figure 5. Both `foldl_du` and `bst_insert_du` update the tree and return void. For such code, programmers *must* consider low-level details such as the representation of the tree and any sharing present. Pawns reflects this by insisting more information is provided in the type signatures (see Section 7). Note that `bst_insert_du` may not compute the same tree as `bst_insert_pure` when there are shared subtrees (see function `sharing_eg`) and thus no local compiler optimisation can convert the pure version to the destructive update version. Importantly, although `list_bst_du` is defined in terms of impure functions, its type and behaviour is that of a pure function, indistinguishable from `list_bst_pure`. The BST returned is an abstract value and cannot be updated further (unless the type signature is changed). We are not aware of other functional programming languages that can encapsulate destructive update in this way.

```
list_bst_du:: Ints -> BST         -- behaves as a pure function
list_bst_du xs =
    *tp = Empty;                  -- allocate mem cell; init to Empty
    foldl_du bst_insert_du !tp xs -- repeatedly insert element

bst_insert_du tp x =              -- returns (), *tp updated
    case *tp of
    Empty ->
        *!tp := Node Empty x Empty  -- insert new node, return ()
    (Node *lp n *rp) ->
        if x <= n then
            (bst_insert_du !lp x) !tp -- update lp (and tp!)
        else
            (bst_insert_du !rp x) !tp -- update rp (and tp!)

foldl_du f y xs =                 -- returns (), y updated
    case xs of
    Nil -> ()                     -- return ()
    (Cons x xs1) ->
        f !y x;                   -- y updated by f
        foldl_du f !y xs1         -- y updated further

sharing_eg:: () -> BST
    subt = Node Empty 42 Empty;
    *tp = Node subt 42 subt;      -- *tp has 3 nodes, subtrees share
    -- bst_insert_pure *tp 42     -- returns BST with 4 nodes
    bst_insert_du !tp 42;         -- inserts 42 into *both* subtrees
    *tp                           -- returns BST with 5 nodes
```

Figure 5. BST creation using destructive update.

We have not performed any benchmarking on non-trivial programs but have written code in various styles and languages (Pawns, C, ML and Haskell) to create a list of 30,000 identical integers and insert them all into a (very unbalanced) BST. Most of the work is in the inner loop of the insert function. For the Pawns destructive update coding, it took around 0.85 s real time on a Dell Latitude E6420 `x86_64` (i7-2620M×4) laptop running Ubuntu 24.04.1 LTS Linux. The pure Pawns coding took around 12.7 s. For some applications, such a slowdown may be a deal breaker. Surprisingly, the destructive update coding in Pawns was the fastest of all the programs we tried. Each Pawns function is compiled to a C function in a very straightforward way, with algebraic data type support provided by the `adtpp` tool [10]. The result for `bst_insert_du` is shown in Figure 6. It becomes rather more verbose and ugly after `adtpp` macros are expanded but `gcc` manages to optimise into very efficient iterative code. Despite several attempts, our fastest hand-written iterative

C code took 1.7 s. It is unwise to draw any strong conclusions from tiny benchmarks but in [8] there is further discussion arguing that the style of destructive updates supported in Pawns can be important for practical applications.

```
void
bst_insert_du(bst* tp, intptr_t x) { // bst pointer tp, unboxed int x
    bst V0 = *tp;
    switch_bst(V0)                   // extract + switch on constructor
    case_Empty_ptr()
        bst V2 = Empty();
        bst V3 = Empty();
        bst V1 = Node(V2, x, V3);   // allocate and initialise Node
        *tp=V1;
    case_Node_ptr(lp, V4, rp)        // assign pointers lp, V4, rp
        intptr_t n = *V4;
        PAWNS_bool V5 = leq(x, n); // leq gets inlined and optimised
        switch_PAWNS_bool(V5)
        case_PAWNS_true_ptr()
            bst_insert_du(lp, x);   // tail recursion, optimised by gcc
            return;
        case_PAWNS_false_ptr()
            bst_insert_du(rp, x);   // tail recursion, optimised by gcc
            return;
        end_switch()
    end_switch()
}
```

Figure 6. Compilation to C, with `adtpp` macros (comments added).

In the second example we use another form of tree, for representing cords. Cords are data types which support similar operations to lists, but concatenation can be performed in constant time. A common use involves building a cord while traversing a data structure then converting the cord into a list in $O(N)$ time, after which the cord is no longer used. Here, we use a simple cord design: a binary tree containing lists at the leaves and no data in internal nodes. Creating a cord from a list plus append and prepend operations can all be performed simply by applying data constructors.

To convert such a cord to a list, a purely functional program would typically copy each cons cell in each list. A C programmer is likely to consider the following more efficient algorithm, which destructively concatenates all the lists without allocating any cons cells or copying their contents. Each `NULL` pointer (`Nil` in Pawns) other than the rightmost one is replaced with a pointer to the first cell of the next list; the first list is then returned (note this updates the cord). This algorithm can be coded in Pawns—see `cord_list` in Figure 7 for the code and Figure 8 for the state of the data structure after the top level call to the auxiliary function `cord_list_a`. The `cord_list` function creates a pointer to an empty list (see `xsp` in Figure 2) and calls `cord_list_a`, which traverses the cord, updating this list (and the cord); then, the list is returned. `cord_list_a` is recursive and is always called with a pointer to a `Nil` (initially `xsp`), which is updated with the concatenated lists from the cord, and it returns a pointer to the `Nil` at the end of the updated list (see `np` in Figure 8). For now, we assume there are only lists of Ints (we will briefly discuss impurity and polymorphism in Section 10.3).

Compared to pure coding, this kind of coding is complicated and prone to subtle bugs and assumptions (thus best avoided except where the added efficiency is important). It may seem that there are several redundant "!" annotations but the Pawns compiler will complain without them. For example, in the first recursive call to `cord_list_a`, with `xc1`, the compiler insists that `xc2` is annotated. Although the analysis performed by the compiler is unavoidably conservative and sometimes results in false alarms, in this case it

is correct. It is possible the lists in the two branches of the cord may share representations, and if this is the case a cyclic list is created and the code does not work! The same can occur if `cord_list_a` is called with `xc` and `np` sharing, instead of `np` pointing to an independent `Nil`. The compiler insisting on extra annotations hopefully alerts the programmer to these subtleties, leading to better documentation and defensive coding to avoid the potential bug.

```
data Cord = Leaf Ints | Branch Cord Cord

-- convert list to cord
list_cord xs = Leaf xs

-- append two cords
cord_app xc1 xc2 = Branch xc1 xc2

-- append list to cord
cord_app_list xc xs = Branch xc (Leaf xs)

-- prepend list to cord
cord_prep_list xs xc = Branch (Leaf xs) xc

-- convert cord to list by efficiently smashing all the lists together -
-- what could possibly go wrong?...
cord_list xc =
    *xsp = Nil;                   -- pointer to empty list of Ints
    np = cord_list_a !xc !xsp;    -- smash all the lists together
    *xsp                          -- return (smashed) list

-- np points to Nil. We smash this list by appending all the lists in xc.
-- We return a ptr to the Nil at the end of the resulting list.
cord_list_a xc np =
    case xc of
    (Leaf xs) ->
        *!np := xs   !xc!xs;   -- smash Nil with xs
        lastp np               -- return ptr to Nil of updated np
    (Branch xc1 xc2) ->
        np1 = (cord_list_a !xc1 !np) !xc!xc2; -- append left subtree
        (cord_list_a !xc2 !np1)      !xc!np   -- append right subtree

-- returns pointer to the Nil of *xsp
lastp xsp =
    case *xsp of
    Nil -> xsp
    (Cons _ *xsp1) -> lastp xsp1
```

Figure 7. Cord operations using destructive update.

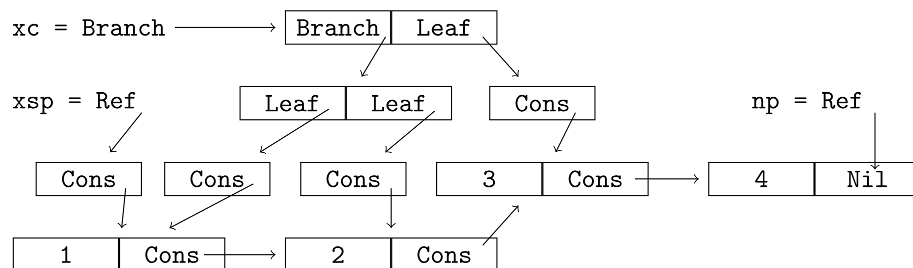

Figure 8. Data layout after calling `cord_list_a`, with `*xsp` now the list `[1,2,3,4]`.

6. Purity and Abstraction

The distinction between pure and impure code can be blurred. For example, some "impure" code can be given "pure" semantics by introducing/renaming variables, adding function arguments, et cetera. Pawns makes a different important distinction, between data structures that are "abstract" (values for which the representation is not important and may not be known by the programmer) versus "concrete" (where the representation, including sharing, may be important and should be understood by the programmer). Only concrete data structures can be updated. Abstract data structures are normally associated with pure code and concrete data structures with impure code, but this is not always the case.

Consider the `lastp` function of Figure 7. It takes a pointer to a list, has no effects and always returns a pointer to `Nil`, so in that sense it is pure (note that pointers themselves are not impure). However, for the destructive update code that uses `lastp`, it matters *which* `Nil` is pointed to in the result. If `lastp` allocated a new memory cell, initialised it to `Nil` and returned a pointer to this `Nil`, the result would be identical from an abstract perspective but the `cord_list` code would not work. Thus, although `lastp` can be considered pure, it must work with concrete data structures. Similarly, impure functions can have abstract arguments and/or results (they cannot update abstract arguments but may update other arguments).

When a data structure is created by applying a data constructor to concrete arguments, the result is concrete. Concrete data structures can become abstract when they are returned from a function (depending on the type signature of the function) or if they are blended with abstract data structures. For example, if an `Empty` subtree of a concrete BST is updated with an abstract BST, or `Node` is applied to one or more abstract BSTs, the result is abstract. Pawns uses the sharing system to keep track of the distinction between abstract and concrete (see Section 7). Pure code such as that in Figure 1 can be written without considering data representation or sharing, but values returned from these functions will be abstract and thus cannot be be updated. Although `lastp` of Figure 7 is pure, the type signature must contain explicit sharing information because we need a concrete list pointer to be returned—the representation is important and the data structure is intended to be updated elsewhere.

7. Sharing Analysis

The Pawns compiler performs *sharing analysis* [12] to approximate how variables share components of their representations and determine what variables may be updated at each point during the evaluation of each function `f`. It relies on knowing what sharing may exist between arguments in calls to `f`, what sharing may exist between arguments and results of functions called by `f`, and what arguments of these functions may be updated. Type signatures in Pawns code have additional information to help this analysis. Specifically, they declare which arguments may be updated, plus a "precondition" stating what sharing between arguments may be present when the function is called and a "postcondition" stating what additional sharing may be present between arguments plus the result when the function returns. As well as the compiler checking there are sufficient "!" annotations, it checks that whenever a function is called the precondition must be satisfied and when a function returns the postcondition must be satisfied. Declaring this additional information is a burden but it forces the programmer to think about sharing in data structures that may be updated, documents sharing for others reading or maintaining the code and helps the compiler conduct analysis to check when destructive update can safely be encapsulated inside pure code and used in the presence of polymorphism. Preconditions can also be used to make code more robust. For example, they can be used to declare that no sharing should exist between the arguments of `cord_list_a` or the functions that build cords: `cord_app`, `cord_app_list` and `cord_prep_list`. Code where such sharing exists will then result in a compiler error message instead of incorrect runtime behavior.

Abstract data structures share with a special pseudo-variable named abstract (there are different versions of this variable for different types, et cetera). For a function that contains

no explicit information concerning sharing, the default precondition and postcondition specify the maximal possible sharing, including sharing with abstract. There is no restriction on calls to such functions (preconditions are always satisfied) but results share with abstract. Code that attempts to update a variable that shares with abstract results in a compiler error. Similarly, passing an abstract data structure to a function that expects a concrete data structure result in an error because the sharing with abstract means the precondition is not satisfied.

A function that has no sharing declared can return a data structure that is concrete within the function because the implicit postcondition simply specifies *possible* (not definite) sharing with abstract. This allows impurity to be encapsulated. For example, within `list_bst_du`, destructive update is used to build a concrete tree but, for code that calls `list_bst_du`, the returned tree will be treated as abstract due to the implicit postcondition. As a general rule in programming, if a possibly shared data structure is updated, the programmer should understand how it has been built and updated, all the way back to the points where each component was created. In Pawns, this must be documented in the code, by explicit declarations whenever it is passed to or returned from a function, and these declarations are checked by the compiler. At some later point we are free to treat it as an abstract value and not concern ourselves with how it is represented or what may share with it, but if this is carried out the value should not be updated further. In Pawns, this is achieved by explicitly or implicitly adding sharing with abstract.

Sharing is declared by augmenting a type signature with a pattern that matches variables with the arguments and result of the function and pre- and postconditions that can use these variables. The pattern can also prefix arguments by "`!`" to indicate the argument may be updated. Preconditions can use the arguments of the function (and `abstract`) to declare the maximal sharing allowed when the function is called. Postconditions can also use the result and declare what additional sharing may be added during the evaluation of the function. The keyword `nosharing` is used to indicate no sharing. Equations and other Pawns code (but not function calls or assignments) can be used to indicate sharing between variables or components of variables. The keyword `inferred` is used to indicate the postcondition is to be inferred from the function definition. This is supported in Pawns for definitions that are pure and contain no function calls (potentially, all postconditions could be inferred but we feel this would detract from the philosophy of Pawns, which makes sharing obvious in the source code wherever it must be understood by programmers). See Figure 9.

The declaration for `list_bst_du` here is equivalent to the declaration in Figure 5 but the sharing with `abstract` is made explicit. For the other `BST` construction code there is no sharing. Integers are atomic; with a more complex data type for elements there would generally be sharing between the list and tree elements and this would need to be declared. Note that, even with no sharing, it needs to be declared, along with the fact that the `BST` is updated, otherwise sharing with `abstract` would be assumed and no update allowed. This applies equally to higher-order arguments such as that in `foldl_du`.

The declarations for the cord code illustrate sharing of variables and their components. Components of variables are discussed further below. For `lastp`, the postcondition states that the result, `np`, and the argument, `xsp`, may be equal (and hence share all components). For `list_cord`, the postcondition states the result, `xc`, may be a `Leaf` whose argument is `xs`, the argument of the function. This is exactly what the function returns but, due to the imprecision discussed below, it means the argument of any `Leaf` data constructor in `xc` may equal `xs`. This more general interpretation is required for `cord_list`. Similarly, for `cord_list_a`, the precondition means a `Leaf` data constructor argument of the cord may equal the list pointed to by the second argument. The precondition of `cord_app_list` prevents it introducing sharing between different lists in a cord, allowing the compiler to reject code that has the bug mentioned earlier (the same should be carried out for other cord construction functions). The postcondition is inferred from the function definition.

```
list_bst_du:: Ints -> BST   -- explicit version of previous code
    sharing list_bst_du xs = t
    pre xs = abstract
    post t = abstract
bst_insert_du:: Ref BST -> Int -> ()
    sharing bst_insert_du !tp x = v
    pre nosharing
    post nosharing
foldl_du::
    (Ref BST -> Int -> ()
        sharing f !xtp x = v
        pre nosharing
        post nosharing
    ) -> Ref BST -> Ints -> ()
    sharing foldl_du f !xtp1 xs = v
    pre nosharing
    post nosharing

lastp:: Ref Ints -> Ref Ints
    sharing lastp xsp = np
    pre nosharing
    post np = xsp
list_cord :: List -> Cord
    sharing list_cord xs = xc
    pre nosharing
    post xc = Leaf xs
cord_list:: Cord -> Ints
    sharing cord_list !xc = xs
    pre nosharing
    post xc = Leaf xs
cord_list_a:: Cord -> Ref Ints -> Ref Ints
    sharing cord_list_a !xc !np0 = np
    pre xc = Leaf *np0
    post np = np0
cord_app_list :: Cord -> List -> Cord
    sharing cord_app_list xc xs = xc1
    pre nosharing  -- If xs shares with lists in xc, list_cord breaks!
    post inferred
```

Figure 9. Type signatures with sharing.

Sharing analysis is unavoidably imprecise but it is conservative, generally overestimating the amount of sharing. Potentially, code may need to have more sharing declared than is actually the case and more variables annotated with "!". For each type, the sharing analysis uses a domain that represents the memory cells that can be used for variables of that type in the running program. For recursive types, the actual number of memory cells can be unbounded, but "type folding" [13] is used to reduce it to a finite number. The domain distinguishes the different arguments of different data constructors but, where there is recursion in the type, the potential nested components are all collapsed into one. For example, for lists, there is a component for the head of the list and another for the tail of the list but, because lists are defined recursively, the head component represents *all* elements of the list (all memory cells that are the first argument of a `Cons` in the list representation) and the tail represents *all* tails.

For cords, there are five components: the two arguments of `Branch`, the argument of `Leaf` and the two arguments of `Cons`. Each left or right branch of a cord is a cord and type folding makes the five components of the branches the same as the top level cord. Thus, for `cord_app_list`, all five components of `xc1` may share with the respective components of `xc`,

along with the two components representing Cons arguments sharing with the respective components of xs. Sharing analysis keeps track of what components may exist for each variable. For example, if a list variable is known to be Nil it has no components at that point in the sharing analysis. Also, note that, for two components to share, they must have the same type and, unless they are pointers, the same enclosing data constructor and argument number. For example, the argument of a Leaf cannot be the same memory location as the second argument of a Cons and sharing analysis respects this distinction. However, we can have a pointer that points to either of these locations, thus sharing analysis treats pointers/refs differently from other data constructors.

8. IO and State Variables

Like destructive update, IO does not fit easily with pure functional programming. Pawns models IO by using a value, representing the state of the world, which is conceptually passed in and returned from all computations that perform IO. Rather than explicitly using an extra argument and a tuple for results, io is declared as "implicit" in the type signature of functions (and nothing is actually passed around). Pawns allows other "state variables" to be defined and (conceptually) passed around in the same way. They are like global variables, but any use or update of them is clear from the source code, and their use can be encapsulated inside a pure interface. In function type signatures, they can be declared as "ro" (read only—as if they are passed in as an argument to the function), "wo" (write only—as if they are initialised/bound by the function and returned) or "rw" (read and written). The io state variable is bound before the main function of a Pawns program is called and all the primitive IO functions have implicit rw io in their type signatures; other state variables must be explicitly bound/initialised before being used. The state variable feature of Pawns is designed with the intention that pure functional semantics *could* be defined (though this has not been formalised). However, calls to functions with implicit arguments/results must be prefixed by ! to highlight the fact than there is more going on in the code than meets the eye, whether or not it is considered pure. State variables are declared like type signatures of functions except they are prefixed with ! and must have a Ref type. They can only be used in code after a wo function has been called or in functions where they are declared implicit in the type signature.

Figure 10 gives a simple example of summing the elements in a BST using a state variable nsum instead of passing additional arguments and results. Although bst_sum behaves as a pure function, as the type signature implies, internally it uses init_nsum to bind/initialise the state variable, which is updated as bst_sum_sv traverses the BST and then its final value is returned. Although bst_sum_sv calls bst_sum (which zeros nsum before traversing the right subtree), the impurity is encapsulated so this does not interfere with the nsum value in the outer computation.

Functions can have multiple state variables declared as implicit arguments with no additional complications. There is no ordering required for the state variables, making some coding simpler compared to mechanisms other languages use for threading multiple kinds of state in a pure way (such as nested monads in Haskell). A disadvantage of using state variables is the code is harder to re-use because it is tied to specific state variables rather than types. State variables and their components can share and be updated in the same way as other Pawns variables. The only additional restriction is that a state variable (or its alias) must not be passed to code where the state variable is undefined (for example, be passed as an argument or returned as a result of a function where the state variable is not declared as an implicit argument). Thus, bst_sum in Figure 10 can return *nsum but not nsum itself, even if the return type and/or the type of nsum was changed. Each state variable is implemented as the address of a static C variable (thus, implicitly passing it around and dereferencing it costs nothing). The single memory location storing the C variable must be protected from "surprise" updates (thus, the integrity of state variables relies on sharing analysis). To allow encapsulated use, the static C variable is saved and restored using a local "auto" variable (stored on the stack) at appropriate points. For example, bst_sum

copies `*nsum` to a local variable at the start of the function and copies it back just before the function returns.

```
bst_sum:: BST -> Int  -- sum of integers in a BST (pure interface)
bst_sum t =
    !init_nsum 0;   -- like nsum = 0
    !bst_sum_sv t;  -- like nsum' = bst_sum_sv t nsum
    *nsum           -- like nsum'

!nsum:: Ref Int  -- declares state variable, nsum

init_nsum:: Int -> ()
    implicit wo nsum  -- binds/initialises/writes nsum
init_nsum n =
    *nsum = n

bst_sum_sv:: BST -> () -- adds all integers in BST to nsum
    implicit rw nsum   -- reads and writes nsum
bst_sum_sv t =
    case t of
    Empty -> ()
    (Node l n r) ->
        *!nsum := *nsum + n;  -- adds n to nsum
        !bst_sum_sv l;        -- adds ints in l (could do same for r)
        *!nsum := *nsum + (bst_sum r) -- uses encapsulated impurity
```

Figure 10. Summing the nodes in a BST using a state variable.

Strictly speaking, state variables are not necessarily single-threaded because components of state variables can be shared with other variables. However, the normal sharing mechanisms of Pawns handles that. If a variable that shares with a state variable is updated, the state variable must be annotated with "`!`" at that point. It may seem that this flexibility destroys the declarative view of IO because we can have code such as `p = *io; !put_char(c); *!io := p` that appears to save and restore the state of the world. However, we can think of `*io` as being a pointer to the "real" state that changes; the pointer remains the same. The type of `*io` is opaque, so there is no way that Pawns code can arrive at the real state or construct a different value of the same type and assign it to `*io`.

9. Polymorphism and Renaming

Sharing in Pawns is not polymorphic to the same extent as types. Similarly, code that uses a state variable is specific to that state variable rather than something more general such as the monad type class in Haskell. For a function such as `foldl`, the second and third arguments do not have identical types declared and Pawns does not allow any sharing to be declared between them. However, for some calls to `foldl`, the types may be identical and we may want to declare sharing between them. In Pawns, this can only be carried out by using a separate function definition that has a more specific type signature with identical types and the sharing declared. Pawns provides a mechanism for renaming groups of functions to simplify this. As an example, Figure 11 shows how the code of Figure 1 code can be duplicated, making it possible to add different type signatures where the sharing is declared and hence the resulting tree can be updated (there is little advantage in having both abstract and concrete versions of these functions, but it does illustrate renaming). The first `renaming` declaration creates definitions of `list_bst_concrete` and `bst_insert_concrete`, by renaming the previous definitions and replacing the call to `foldl` by a call to `foldlBST`. An explicit definition of `foldlBST` could be included but we here simply use another `renaming` declaration. Type signatures are needed for all

three functions (for brevity, we only include one). Renaming can also be used as a less abstract alternative to higher-order code, and for producing code with the same structure but with different state variables. For example, we can code a version of `map` that uses `io` and rename it to use other state variables as needed (this is the Pawns equivalent of using Haskell's `mapM`).

```
renaming
    list_bst_concrete = list_bst_pure
    bst_insert_concrete = bst_insert_pure
    with
    foldlBST = foldl

-- same effect as just deleting the "with" above
renaming
    foldlBST = foldl

-- also need type signatures for list_bst_concrete and foldlBST
bst_insert_concrete:: BST -> Int -> BST
    sharing bst_insert_concrete xt x = xt1
    pre nosharing
    post xt1 = xt
```

Figure 11. Renaming of function definitions.

10. Complications

Combining pure functional programming with destructive update and other impurity is not simple! The design of Pawns aims to support high-level pure functional programming plus low-level imperative programming with as much flexibility as possible while avoiding unsafe operations (such as dereferencing `NULL` pointers) and "surprises" (code with effects that are obscure). Here, we briefly mention some of more complicated issues and how they are dealt with in Pawns, without too much technical detail.

10.1. Foreign Language Interface

The one feature of Pawns where there is no attempt to guarantee safety is the foreign language interface. Pawns compiles to C and provides a simple and flexible interface to C, which has many unsafe features. Each Pawns function compiles to a C function and Pawns allows the body of a function definition to be coded in C, but for such code there can be no guarantees of safety or lack of "surprises". It is up to the programmer to ensure the C code is safe and compatible with the Pawns type signature. For example, Figure 12 gives the implementation of `put_char` defined in terms of `putchar` in C. The use of the `io` state variable in the type signature ensures that the code can only be used in a context where the side-effect is clear and purely functional semantics could be defined. Similarly, it only requires a few lines of code to interface Pawns to the C standard library pseudo-random number package in a way that can be encapsulated and given purely functional semantics, using a state variable—see Figure 12 for the type signatures. It is also very easy to support arrays via the C interface. The current code has an option for omitting bound checks (thus gaining C-like efficiency but sacrificing safety).

Most foreign language interfaces only allow basic unstructured types to be passed. However, the Pawns compiler uses the `adtpp` tool [10], which generates C macros for manipulating the algebraic data types defined in the program—see Figure 6. These macros can also be used in hand-written C code to both operate on a `BST` that was created by Pawns code, and create a `BST` that is passed back to Pawns code. Dynamic memory management is often particularly difficult across language boundaries but is made very easy in Pawns by using the Boehm–Demers–Weiser conservative garbage collector [14].

```
put_char: Int -> ()
    implicit rw io
put_char i = as_C "{putchar((int) i);}"

-- pseudo-random number sequence interface
init_random:: int -> () -- initialize sequence with a seed
    implicit wo random_state
random_num:: () -> int  -- return next number in sequence
    implicit rw random_state
```

Figure 12. C interface.

10.2. Higher-Order Programming

There are two complications involving higher-order code: type checking and partially applied functions (closures). Type checking is made more complicated because each "arrow" type has additional information concerning sharing, destructive update and state variables. Pawns allows some latitude when matching the type of arguments to higher-order functions with the expected type that is declared. The arguments are allowed to have less destructive update, less sharing in postconditions, more sharing in preconditions and some variations in what state variable operations are declared (for example, `ro` is acceptable where `rw` is declared). The intention is to allow as much flexibility as possible while guaranteeing safety.

Pawns allows functions to be applied to fewer than the declared number of arguments, resulting in closures being constructed/returned. Closures can be passed around like other data and later applied, leading to function evaluation. The arguments inside closures can share with other data structures and, hence, they can potentially be updated—a source of surprises in many programming languages. In Pawns, variables that are closures must be annotated with "`!`" wherever they may be updated, just like other variables. Note that certain equivalence laws that hold for pure functional programming (such as "eta-equivalence") do not hold when closures may be updated.

The patterns used for declaring sharing and destructive update can have additional arguments, representing the arguments of closures, so they can be declared as updated and appear in preconditions and postconditions. In most cases, declaring sharing and update of closure arguments is not carried out explicitly as it is inferred by the compiler. For example, with `cord_list_a:: Cord -> Ref Ints -> Ref Ints`, the explicit sharing is associated with the rightmost (innermost) arrow of the type, where both arguments have been supplied. The compiler infers sharing and destructive update for the other arrow, including the sharing between the first argument of `cord_list_a` and the closure that would be returned if only one argument is supplied. Because there is no computation performed before returning the closure, there is no destructive update possible at this point. State variable declarations are similarly inferred for non-rightmost arrows. Inferred declarations can be overridden by explicit declarations in the code.

10.3. Polymorphism and Type Safety

Mixing polymorphic types with destructive update can result in unsafe operations if it is not carried out carefully. Consider the code in Figure 13. The variable `xsp` is bound to a pointer to `Nil`, a list of *any* type. Without destructive update, this can be safely used where pointers to lists of integers and pointers to lists of binary search trees are expected (the type can be instantiated to either of these without problems). However, if the variable is updated to be a non-empty list of integers, the code is not type safe—an integer may appear where a tree is expected. Other functional languages solve the type safety problem by imposing restrictions on code that has refs (and thus may perform updates), such as the "value restriction" in ML [11]. In Pawns, refs to arguments of data constructors can be created anywhere but, because the source code explicitly notes where variables can be updated, the problem can be solved in a more flexible way.

Where a Pawns variable with a polymorphic type may be updated (by an assignment statement or an impure function call), type variables may become more instantiated during type checking. For example, at the point where `xsp` is passed to `int_fn` in Figure 13, its previous polymorphic type (`Ref (List t)`) is instantiated to `Ref (List Int)`. The type of `xsp1` is also similarly instantiated—the two variables share their representations and their types shared the same type variable, `t`. The subsequent call to `bst_fn` then results in a type error. Pawns treats all variables created with polymorphic types as live throughout the whole function, so the `!` annotation on `xsp` is required even if `xsp` is never used again, alerting readers of the source code to a subtlety. The compiler also prints a warning when types are further instantiated. Warnings can be avoided by adding explicit casts, as shown in the second example of Figure 13.

```
    *xsp = Nil;       -- Nil is a list of any type
    xsp1 = xsp;       -- xsp1 has the same polymorphic type as xsp
    ys = (int_fn !xsp) !xsp1; -- int_fn accepts a ref to a list of ints
    -- Now *xsp (and *xsp1) may be a non-empty list of ints!
    zs = bst_fn *xsp1; -- OOPS! bst_fn accepts a ref to a list of BSTs

cord_list xc =
    *xsp = Nil::Ints;           -- instantiate list type explicitly
    np = cord_list_a !xc !xsp;  -- xsp has a monomorphic type
    *xsp
```

Figure 13. Potential violation of type safety.

10.4. A Formal Proof of Safety

Currently, there is no formal proof of safety for Pawns. Despite our confidence that the overall approach of Pawns is sound, more formality would be beneficial but it is technically quite challenging. First, some formalisation of the operational semantics of Pawns would be necessary, including memory use. Second, the type checking algorithm would need to be formalised more. For handling higher-order code, some simplification might make an initial proof easier, such as omitting state variables and more complex matching of sharing information. Third, there is mutual dependence between type checking and sharing analysis. The way type checking instantiates type variables assumes the code includes annotations where polymorphic variables may be updated. Annotations are checked during liveness and sharing analysis, which is carried out *after* type analysis. The order cannot be changed because the sharing abstract domain requires type information. For the code in Figure 13, if either "`!`" annotation was omitted an error would be detected during sharing analysis rather than type analysis. The (unproven) conjecture is code that passes both type checking and sharing analysis is safe, but untangling this dependence adds to the challenge of a fully formal proof. The sharing analysis algorithm itself is quite complicated. It uses a simplified "core" language and seeks to establish a condition which could be used in an inductive proof of correctness. The published version [12] is slightly outdated. It uses a less expressive abstract domain than is currently used and a more restricted version of type checking that did not instantiate the types of polymorphic variables where they could be updated, so more type casts were required in Pawns code.

11. Further Work

As mentioned above, there is work to be carried out on formalising the type system and proving safety, etc. There are three enhancements to Pawns we are particularly keen to implement. The first is to the specification of destructive update. Both in statements and type signatures, variables can be declared mutable or not. We are currently working on a more refined approach where programmers can declare that only certain components are mutable. For example, for BST insertion, it is very be desirable to express the fact that only the tree nodes are updated, allowing a mutable BST containing abstract values in the nodes

(this cannot be performed with imprecise mutability declared because abstract values must not be updated). We have developed a small language to specify which data constructor arguments in a term can be mutable. In the BST case, we use "`...Node ! ? !`", where "`!`" indicates mutability, "`?`" indicates no mutability (so an abstract value could be used) and "`...`" indicates any number of outer data constructors (which must all be `Node` for a BST). For function `cord_list_a`, the specification for references to cords is "`Ref !(...Cons ? !)`", indicating an outermost ref data constructor with a mutable argument and some path down to a cons cell whose second argument is also mutable. The compiler now supports more precise description of destructive update in type signatures and statements, using this language.

The second enhancement is a new method of specifying sharing. Currently, sharing declarations use Pawns code, without function calls or assignment, but allowing extra flexibility such as defining a variable more than once. For example, `x=y; x=z` is allowed. Function calls and assignment are disallowed because we felt they make the sharing too subtle. A more direct approach would be to invent a small language for naming components of data structures (similar to what we have carried out for more precise destructive update) and use equality operators between them. Being able to express that a term is acyclic and/or there is no sharing between different sub-terms would also lead to greater precision. The current way of defining sharing is a practical one, allowing us to re-use the existing sharing analysis code rather than invent a new language.

The third enhancement is to support the creation of pointers to arguments of data constructors at the same time as the data constructors are created; for example, we could use `x = Cons (*headp=1) (*tailp=Nil)` to make `x` the list `Cons 1 Nil` and at the same time create variables `headp` and `tailp` that point to the two arguments of `Cons`, respectively. Currently this requires a separate switch statement.

There are also several features of other functional programming languages such as existential types and type classes that it would be good to consider for Pawns. Integrating features that make types more abstract, such as type classes, with sharing would pose a challenge because at least our current view of sharing is tied closely to the concrete representation of values. Moving from the current proof of concept to a more practical language may be worthwhile but it would take a significant amount of (not very interesting) work. Some of the features of Pawns could also be incorporated into other languages. For example, reference patterns could be incorporated into Discus easily. The way Pawns supports mutable algebraic data types and state variables would also be beneficial for imperative languages.

12. Related Programming Languages

We now briefly discuss some of the key language design issues surrounding Pawns and relate them to some other languages. There are *many* imperative languages designed to address various safety, security and performance issues. Some have features such as option types (a very simple instance of an algebraic data type) instead of Hoare-style pointers that can be NULL, data structures that are protected from being updated and the like. The adtpp tool [10] supports safe and efficient algebraic data types for C, with the update of data constructor arguments and a form of reference patterns; it was developed by us in parallel with Pawns and is used in the Pawns compiler. However, to avoid this section becoming large and/or superficial, we restrict the discussion to languages for which purity (and related concepts) is an issue; mostly functional and logic programming languages. A more detailed and extensive discussion is contained in [9], which describes the Mars language, and [8], which describes the Disciple language (now called Discus). Mars is very different from Pawns in many ways but both languages adhere to the principle that all dependencies and effects should be obvious from the code.

We first discuss the update of data structures that are not shared. Many declarative languages are designed and/or implemented so destructive update can be performed on such data structures. This is particularly important for arrays, since destructively updating

an element of an array of size N takes $O(1)$ time whereas creating a new copy takes $O(N)$ time and space. Various language features and compiler analyses (discussed below) have been used to ensure or check that data structures are "single-threaded" through the computation so sharing is limited or avoided. That is, if a data structure is modified, the new version of the data structure will typically be used but the old version will never be used again, so it does not matter if it is destroyed. There may be variables whose value is (conceptually) the old version but these variables/references are "dead"—they have no further occurrences and can be ignored. This is particularly important for modelling IO in a pure way, especially in non-strict functional languages where the evaluation order may be hard to determine. Conceptually, a "state of the world" is passed through the computation and after some IO operation has been performed the previous state no longer exists.

In Pawns, ignoring state variables for now, "`!`" is only required when "live" variables are updated. Figure 14 gives a simple example showing how the update of a single-threaded data structure raises no concern: where `i2` is bound, `i1` is no longer used so "`!`" is not required. When `i2a` is bound to `i2`, the data structure is not single-threaded because `i2` is also used later (as is `i2a`). However, when the next update is performed, there are no live references remaining so "`!`" is still not required. For both these increment operations the code behaves as if the pure function `inc_pure` was used instead of `inc_du`. Annotations are only needed when an updated variable, or something that shares with it, is used later (the bindings of `i4` and `i5`).

The type system of Clean (https://wiki.clean.cs.ru.nl/Clean, accessed on 13 November 2024) [15] enforces single threading by using uniqueness types, based on substantial theoretical work on linear types and linear logic [16]. Variables with linear types can be used only once; thus, the compiler is free to emit code that updates them rather that reconstructing the data structure (replacing code like `inc_pure` with `inc_du`, at least for arrays—even with linearity, updating other data structures safely has proved more challenging). Linear types have also been incorporated into an experimental extension to Haskell [17]. A similar mechanism is used in the logic programming language Mercury (http://mercurylang.org/, accessed on 13 November 2024) [18]. Program units are predicates rather than functions and each argument has a type and (possibly more than one) "mode", specifying whether the argument is an input or an output. The modes "unique output" and "destructive input" signal that an output variable should only be used once (and may be destroyed/updated at that point).

Mars [9] is an imperative programming language but it is designed with "interface integrity" in mind; that is, when an execution unit such as a function is invoked, all dependencies and effects should be obvious from either the function call or the declaration of the function. Arguably, if a language has interface integrity it has the main benefit of purity, whether or not the code is considered pure. Wybe (https://github.com/pschachte/wybe/, accessed on 13 November 2024) also follows this principle, but is influenced by the logic programming paradigm in that many programming constructs define predicates/relations between values and the distinction between inputs and outputs depends on where annotations appear. A variable with no annotation is an input, a variable prefixed with "?" is an output (it is assigned a value) and a variable prefixed with "`!`" is both an input and an output. Both `?x = y` and `y = ?x` assign the value of `y` to `x` whereas `x = y` tests whether the two values are equal—three different modes for the equality relation.

Mars and Wybe support data constructors with named fields/arguments and syntax that *seems to* allow the destructive update of fields, for example `myVar.someField = newVal`. However, the semantics is that the structure is copied, with all fields the same as previously except that `someField` is assigned `newVal`, and the new structure is assigned to `myVar`. This copying semantics prevents the update of shared structures, so the assignment does not affect any other variables that may have shared with the original structure. If compiler analysis concludes there are no other live references to the structure, then destructive update can be used instead of copying. Although Pawns, Mars and Wybe are designed with interface integrity in mind, they take almost opposite approaches to update. Mars

and Wybe allow the update of variables but not data structure arguments (semantically at least); Pawns is the opposite. For code that seems to perform the update of structures, Mars and Wybe actually use the pure (copying) alternative, unless the compiler determines destructive update has the same result. In Pawns, destructive update is always carried out but, if the compiler determines copying has the same result, no "`!`" is needed so the code appears pure (see Figure 14). To detect when destructive update can be used instead of copying, Mars and Wybe perform sharing analysis of the kind used in Pawns. It is also used in Mercury [19] for compile time garbage collection and structure re-use, in Prolog [20] for similar optimisations and in Koka (https://koka-lang.github.io/koka/, accessed on 13 November 2024) (see below).

```
    ...
    *i1 = 1;
    i2 = inc_du i1;        -- single threaded (i1 is not used later)
    i2a = i2;              -- i2a aliases/shares with i2
    i2b = i2a;             -- i2b aliases/shares with i2 and i2a
    i3 = inc_du i2;        -- no ! (i2, i2a and i2b are not used later)
    i4 = inc_du !i3;       -- i3 is used later so ! is needed
    i3a = i3;              -- i3a aliases/shares with i3
    i5 = (inc_du i3) !i3a; -- i3a is used later so ! is needed
    i6 = inc_du i3a
    ...

-- increment int destructively; returns original ref
inc_du:: Ref Int -> Ref Int
    sharing inc_du !prt = r
    pre nosharing
    post res = ptr
inc_du prt val =
    *!prt := *ptr + 1;
    ptr

-- returns the same value as inc_du but is pure
inc_pure:: Ref Int -> Ref Int
inc_pure(ptr) =
    *newptr = *ptr + 1; -- allocates new memory cell
    newptr              -- returns pointer to new memory cell
```

Figure 14. Update of a (sometimes) single-threaded data structure.

Haskell uses monads [21] (from category theory) to thread IO through a computation. There is no data structure or variable that represents the state of the world, simply a "phantom" value that is conceptually passed around but cannot be accessed. Monads can also be used to thread arrays and other values through a computation, allowing destructive update primitives to have a pure semantics. For such code, Haskell provides the "do" notation, an extensive syntactic sugar that necessitates major code modification when state threading is added to a previously stateless computation. It is common for designers of declarative languages to model IO by conceptually threading a state of the world through the execution, then invent some syntactic sugar to make this threading less cumbersome and ensure the compiler can optimise it away. The same mechanism can be used for threading other states, though optimising non-IO state threading away so copying and/or passing extra arguments and results may depend on the state not being shared. In Mercury, the definite clause grammar notation of Prolog can be used to make the last two arguments of predicates implicitly thread state and this syntactic sugar has been extended to allow multiple states to be passed relatively easily. The Haskell "do" notation is particularly cumbersome when access to and modification of multiple forms of state is required. Pawns state variables give the convenience of global variables but their use can be encapsulated

and is made more obvious in the code. They have the advantage of simplicity, particularly when there are multiple forms of state, but make re-use of code (particularly higher-order functions) more difficult than some other approaches. Wybe supports "resources", which are very similar to state variables: functions/procedures that use them must declare them in the type signature, including whether they are read, written or both, and calls to such functions must be prefixed by "`!`".

Koka uses effect typing (based on category theory), where effects such as exceptions, destructive update and IO are attached to types. For example, getchar has return type `console int`, indicating it returns an integer but also may have an effect on the console, by performing IO (similar to the monadic type `IO int` in Haskell). Koka has mutable local variables (with some restrictions to avoid them escaping their scope) and uses a combination of compile time analysis and runtime reference counting to enable the re-use of structures that have a single reference (rather than allocating new structures). Thus, single-threaded data structures can be destructively updated, with the minor overhead of checking the reference count. One advantage of this approach is that the same code can be used for both unique and shared structures. Also, the memory used by one data constructor can be re-used by another, if it is the same size.

An important distinguishing feature of Pawns is how it allows the destructive update of *shared* structures. Such code is inevitably subtle but there are important algorithms that require this capability. Pawns stands alone in the way the destructive update of shared structures is made obvious in the source code and can be encapsulated. ML introduced "refs" to support it and some other functional languages have adopted similar constructs. Incorporating such a feature into a non-strict language is more challenging, but Haskell does so using monads (the `STRef` library) so the store state can conceptually be threaded through the computation and the feature can be considered pure. Koka has a ref type and uses effects of the form `st`<*h*>, where *h* represents a (mutable) heap that is associated with the data type. Supporting a mutable `ref` or `STRef` type essentially allows the argument of a single data constructor (that used in the ref type) to be updated, but that data type must be explicitly included in user-defined high-level type definitions. This can lead to multiple variants of data types, depending on which components need to be updated, and less efficient data representation, since there is an extra level of indirection. Furthermore, including refs in a data structure so one function can update it allows *all* functions to update it—the impurity/mutability "leaks" into surrounding code and it is similar to the situation in imperative languages. The only way of ensuring it will not be updated further is to re-build the data structure using a different type that does not contain refs. The same occurs with other mutable data structures in Haskell: if a mutable array is created (inside a monad), all functions that the array is passed to can potentially update it.

Haskell does support `unsafeFreeze` and `unsafeThaw` functions that can convert between mutable and immutable array types without copying data, but it is up to the programmer to ensure they are used in a way that avoids surprises such as immutable arrays actually being updated. Similarly, it has functions such as `unsafePerformIO` that can result in code impurity. Mercury also allows code to have "promises" made by the programmer concerning purity, which are not checked by the compiler. So far, we have not felt the need to have such features in Pawns, though admittedly we have not written much Pawns code so it is possible that for larger projects imprecision in the analysis may have worse consequences than a few additional "`!`" annotations being needed.

Discus [8] is by far the most similar language to Pawns in terms of update capabilities. It is a functional language that supports the update of the arguments of all data constructors rather than a distinguished "ref" data constructor, so additional redirection and types are not required. Discus allows the creation of refs to named fields of records (data constructor arguments) using the syntax `myVar#someField` and the equivalent of Pawns `*myRef = val` is written `myRef #= val`. Discus adds memory "region" information [22] to the type system. For example, given a set of variables which are lists, the region information partitions the set according to which region their cons cells occupy. There is a distinction between reading

and updating data in each region, so the types can be used to determine that a function does not update a data structure. However, there is "leakage" of mutability information in a similar way to that with refs. In the Discus equivalent of `list_bst_du`, the region containing the tree nodes must allow updates and that remains when the tree is passed to other functions, so those functions may update the tree (similarly to the way the `STRef` monad in Haskell and `st` effects in Koka propagate mutability beyond the function where it is needed). Type information, including the update of regions, is rather daunting in its complexity but it can be inferred. However, because the update of function arguments can be inferred, it is not necessarily obvious from the source code as it is in Pawns.

Region information is less expressive than sharing information and must be transitive: if `x` is in the same region as `y` and `y` is in the same regions as `z`, `x` also must be in the same region as `z`. All arguments of the same data constructor must also be in the same region. Non-transitive sharing information is important for preconditions and postconditions. For example, in `cord_app_list`, we had the postcondition `xc = Branch xc0 (Leaf xs)`, meaning `xc` may share with `xc0` and `xs`, but `xc0` and `xs` do not share. The fact that `xc0` and `xs` do not share is the precondition of the function, which is important for the correctness of the cord code. This precondition cannot be expressed with region information. The distinction between different arguments of data constructors can be important also. In Pawns, it is possible to have two pointers to different arguments of the same data constructor, only one of which is updated and annotated as such. Similarly, they may have quite distinct sharing (they may not even have the same type).

13. Summary of Contributions

Pawns supports many of the attractive features of pure functional programming while also supporting encapsulated impure operations such as the destructive update of shared data structures. The main novel features it contributes are as follows:

- Consideration of sharing is part of the language, not simply the implementation.
- All effects, including effects due to sharing, are obvious from the source code.
- Reference patterns create references to data constructor arguments.
- There is a distinction made between concrete and abstract data structures.
- Destructive update can be encapsulated by returning abstract data structures.
- State variables implement another form of encapsulated effects.
- Renaming of functions can be used instead of polymorphism and higher-order code.

14. Threats to Validity

Essentially, our claim is the novel features of Pawns make it a valuable contribution to the programming language landscape. The value of a programming language is largely dependent on how many people use it, and this is driven by a range of factors extrinsic to the language. There is also positive feedback—a language with a larger user base will generally be supported better, have more libraries developed, be taught more to students and be chosen for more projects, increasing the user base further. Pawns will never be in this category. However, a language can also have intrinsic value and have features that are influential. For example, although nobody uses Simula [23] today, it is generally considered to be the first language with object-oriented features, which are now widespread. Similarly, the original version of Lisp [24] is not used today but it spawned many variants, including Common Lisp, Scheme, Racket and Clojure. Furthermore, it was the first language to introduce both anonymous functions and garbage collection, which now contribute greatly to the productivity of many programmers using many very different languages. There are also language features such as algebraic data types, introduce by Hope [25], which are used in many functional programming languages but are (arguably (https://lee-naish.github.io/papers/adtpp/adtrant.html, accessed on 13 November 2024)) underutilised and far more widespread use would be very beneficial.

The main threats to the validity of our claim are external. It is possible (quite likely in fact) that nobody will use or be influenced by Pawns or any of its features. This is a

major threat to all programming language development that is not backed by significant resources. Publishing this paper reduces the threat somewhat. Declaring sharing could also turn out to be considered too much of a burden for most programmers. This could possibly be alleviated by devising different ways of expressing sharing and/or the compiler could make suggestions for valid sharing declarations based on inferred sharing information. The error messages currently provided by the compiler also provide guidance. To a lesser extent, declaring destructive update could prove unpopular.

The main internal threat is a major flaw in the sharing and/or type analysis algorithms that cannot easily be fixed. The sharing algorithm is very complex and there have been numerous bugs detected during development but all have be relatively easy to fix. A version of the sharing algorithm that used a slightly less precise abstract domain was peer-reviewed. The type analysis is also complex and has been modified during development so fewer type casts are required in code. Precise information about where data structures may be updated allows for a degree of flexibility in type analysis. It would also be possible to have two passes of type analysis in the compiler, the first to provide the domain for sharing analysis and the second to address type safety in the presence of polymorphism.

The renaming of functions is not affected by the threats above but in itself is not a big contribution. The motivation for adding it to Pawns was that both sharing and state variables are not polymorphic to the same extent as types, so it was desirable to have two separate declarations (and names) for essentially the same function. It may also be an attractive alternative to higher-order functions for some people. State variables are also less affected by these threats. A version of state variables could be used instead of global variables in an imperative language as long as no aliases for a state variable can escape from the context where it is defined. For example, taking the address of a state variable could be disallowed to avoid aliasing (full sharing analysis is not required).

15. Conclusions

There are important algorithms which rely on the destructive update of shared data structures, and these algorithms are relatively difficult to express in declarative languages and are typically relatively inefficient. The design of Pawns attempts to overcome this limitation while retaining many of the advantages of a typical functional programming language, such as algebraic data types, parametric polymorphism and higher-order programming. Pawns supports the creation of pointers to arguments of data constructors, and these pointers can be used for the destructive update of shared data structures. There are several features which restrict when these effects can occur and allow them to be encapsulated, so the abstract declarative view of some functions can still be used, even when they use destructive update internally.

Type signatures of functions declare which arguments are mutable and, for function calls and other statements, variables are annotated if it is possible that they could be updated at that point. In order to determine which variables could be updated, it is necessary to know what sharing there is. Functions have pre- and postconditions which describe the sharing of arguments and the result when the function is called and when it returns. To avoid having to consider the sharing of data structures for all the code, some function arguments and results can be declared abstract (this is the default). Reasoning about code which only uses abstract data structures can be identical to reasoning about pure functional code, as destructive update is prevented. Where data structures are not abstract, lower-level reasoning must be used—the programmer must consider how values are represented and what sharing exists. The compiler checks that declarations and definitions are consistent, allowing low-level code to be safely encapsulated inside a pure interface. Likewise, the state variable mechanism allows a pure view of what are essentially mutable global variables, avoiding the need for source code to explicitly give arguments to and extract results from function calls. Analysis of sharing is also used to ensure the use of state variables can be encapsulated and to ensure the safety of code that uses destructive update of polymorphic data types.

Although Pawns is still essentially a prototype, and is unlikely to reach full maturity as a "serious" programming language, we feel its novel features add to the programming language landscape. They may influence other languages and help combine the declarative and imperative paradigms, allowing both high-level reasoning for most code and the efficiency benefits of the destructive update of shared data structures.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The Pawns source code is freely available on GitHub: https://github.com/lee-naish/Pawns (accessed on 13 November 2024).

Acknowledgments: The design of Pawns has benefitted from discussions with many people. Bernie Pope and Peter Schachte particularly deserve a mention. I also appreciate the extensive comments provided by one of the reviewers.

Conflicts of Interest: The author declares no conflicts of interest.

References

1. Naish, L. An Informal Introduction to Pawns: A Declarative/Imperative Language. 2015. Available online: https://lee-naish.github.io/papers/pawns/ (accessed on 13 November 2024).
2. Jones, S.P. (Ed.) *Haskell 98 Language and Libraries: The Revised Report*; Cambridge University Press: Cambridge, UK, 2002; p. 277. Available online: https://haskell.org/ (accessed on 13 November 2024).
3. Kernighan, B.W.; Ritchie, D.M. *The C Programming Language*, 2nd ed.; Prentice Hall Professional Technical Reference; Prentice Hall: Englewood Cliffs, NJ, USA, 1988.
4. Wadsworth, C. *Semantics and Pragmatics of the Lambda-Calculus*; University of Oxford: Oxford, UK, 1971.
5. Warren, D.H.D. *An Abstract Prolog Instruction Set*; Tecnical Note 309; SRI International: Menlo Park, CA, USA, 1983.
6. Taylor, A. Parma—Bridging the Performance GAP Between Imperative and Logic Programming. *J. Log. Program.* **1996**, *29*, 5–16. [CrossRef]
7. Arden, B.W.; Galler, B.A.; Graham, R.M. An algorithm for equivalence declarations. *Commun. ACM* **1961**, *4*, 310–314. [CrossRef]
8. Lippmeier, B. Type Inference and Optimisation for an Impure World. Ph.D. Thesis, Australian National University, Canberra, Australia, 2009. Available online: https://benl.ouroborus.net/papers/2010-impure/lippmeier-impure-world.pdf (accessed on 13 November 2024).
9. Giuca, M. Mars: An Imperative/Declarative Higher-Order Programming Language with Automatic Destructive Update. Ph.D. Thesis, University of Melbourne, Melbourne, Australia, 2014.
10. Naish, L.; Schachte, P.; MacNally, A. Adtpp: Lightweight efficient safe polymorphic algebraic data types for C. *Softw. Pract. Exp.* **2016**, *46*, 1685–1703. [CrossRef]
11. Milner, R.; Tofte, M.; Macqueen, D. *The Definition of Standard ML*; MIT Press: Cambridge, MA, USA, 1997.
12. Naish, L. Sharing analysis in the Pawns compiler. *PeerJ Comput. Sci.* **2015**, *1*, e22. [CrossRef]
13. Bruynooghe, M. Compile time garbage collection or how to transform programs in an assignment free languages into code with assignments. In Proceedings of the IFIP TC2/WG2.1 Working Conference on Program Specification and Transformation, Bad-Tölz, Germany, 15–17 April 1986.
14. Boehm, H.J.; Weiser, M. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exper.* **1988**, *18*, 807–820. [CrossRef]
15. Plasmeijer, R.; van Eekelen, M. *Concurrent Clean Language Report, Version 2.1*; University of Nijmegen: Nijmegen, The Netherlands, 2002.
16. Girard, J.Y. Linear logic. *Theor. Comput. Sci.* **1987**, *50*, 1–101. [CrossRef]
17. Bernardy, J.P.; Boespflug, M.; Newton, R.R.; Peyton Jones, S.; Spiwack, A. Linear Haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* **2017**, *2*, 5. [CrossRef]
18. Somogyi, Z.; Henderson, F.; Conway, T.C. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *J. Log. Program.* **1996**, *29*, 17–64. [CrossRef]
19. Mazur, N.; Ross, P.; Janssens, G.; Bruynooghe, M. Practical aspects for a working compile time garbage collection system for Mercury. In *Logic Programming, Proceedings of the 17th International Conference, ICLP 2001, Paphos, Cyprus, 26 November–1 December 2001*; Codognet, P., Ed.; Springer: Berlin/Heidelberg, Germany, 2001; Volume 2237, pp. 105–119. [CrossRef]
20. Mulkers, A. *Live Data Structures in Logic Programs, Derivation by Means of Abstract Interpretation*; Springer: Berlin/Heidelberg, Germany, 1993. [CrossRef]
21. Peyton Jones, S.; Wadler, P. Imperative functional programming. In Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, SC, USA, 10–13 January 1993; pp. 71–84.
22. Talpin, J.P.; Jouvelot, P. Polymorphic type, region and effect inference. *J. Funct. Program.* **1992**, *2*, 245–271. [CrossRef]

23. Dahl, O.J.; Nygaard, K. SIMULA: An ALGOL-based simulation language. *Commun. ACM* **1966**, *9*, 671–678. [CrossRef]
24. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* **1960**, *3*, 184–195. [CrossRef]
25. Burstall, R.M.; MacQueen, D.B.; Sannella, D. HOPE: An experimental applicative language. In Proceedings of the LISP Conference, Stanford, CA, USA, 25–27 August 1980.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.