An informal introduction to Pawns: a declarative/imperative language

Lee Naish

University of Melbourne, Melbourne 3010, Australia
lee@unimelb.edu.au,
http://people.eng.unimelb.edu.au/lee/

March 14, 2015

Abstract. Pawns is yet another programming language under development which attempts to combine the elegance of declarative programming with the algorithmic expressive power of imperative programming. Algebraic data types can be defined and viewed as decriptions of high level values which can be manipulated in declarative ways. The same type definitions can also be viewed at a much lower level, involving pointers to possibly shared data structures which can be destructively updated. Pawns programs contain annotations and declarations which indicate whether the high or low level view should be used in different program components and what variables may be updated at each point. This makes all effects obvious and allows efficient imperative code to be encapsulated within a declarative interface, with "purity" of functions and consistency of annotations and declarations being guaranteed by the compiler.

Keywords: functional programming language, destructive update, mutability, effects, algebraic data type, sharing analysis

1 Introduction

This paper informally describes Pawns, a programming language which is currently under development. We assume the reader is familiar with Haskell and C. Many aspects of Pawns have not yet been finalised, but we describe some of the core features which are novel. Pawns supports both declarative and imperative styles of programming. Declarative programming languages allow us to write and reason about code at a very high level. We can define data types which precisely describe the values we want to compute with, and these can be manipulated without us considering how or where the values are stored. Many programming errors, such as those related to memory management, `NULL` pointers, etc are impossible to make. However, there are some algorithms which cannot be expressed easily and implemented efficiently in purely declarative languages. Pawns supports many of the advantages of high level declarative programming but also allows low level imperative programming so that these algorithms can be implemented efficiently.

We believe that being able to view code at different levels of abstraction is a very important part of good programming, whatever the language. For example, an abstract data type can and should be thought of as representing and manipulating values in some abstract domain when viewed from outside its interface, but when viewing the implementation, inside the interface, a lower level understanding is essential. In a similar way Pawns encourages many functions to be viewed in a purely declarative way from outside their interface, but internally they may use imperative style programming where data must be viewed at a much lower level, being stored in memory cells which can have pointers to them. The main contribution of Pawns is how it supports these different levels of abstraction and makes it clear which view is appropriate at each point.

Pawns has the look and feel of a functional programming language. We present it using Haskell-like syntax[1] and (ignoring superficial syntactic differences) Haskell code using features such as polymorphism, curried functions and higher order programming are fully supported. Existential types and type classes are not currently supported but could easily be added in the future. However, it uses strict evaluation and this cannot easily be changed. The core ideas of Pawns could equally be applied to add encapsulated impure features to a logic programming language. Alternatively, they could be used in an imperative or object oriented language to support algebraic data types and make data-flow dependencies between different parts of the code more obvious.

The rest of this paper is structured as follows. In Section 2 we discuss the relationship between declarative and imperative programming in a little more detail to help motivate the core contributions of Pawns. In Section 3 we describe how algebraic data types can be viewed at different levels of abstraction. In Section 4 we describe the basic mechanisms of how values are constructed, deconstructed and destructively updated in Pawns. In Section 5 we describe additional constraints on updating values which are imposed by the compiler. In Section 6 we present some example programs and also discuss other core features of Pawns. In Section 7 we give a brief overview of the analysis and checking done by the Pawns compiler. In Section 8 we discuss mixing high and low level code and reasoning about partial correctness. In Section 9 we discuss IO and features which support concise threading of state information. In Section 10 we discuss polymorphism. In Section 11 we discuss higher order programming. In Section 12 we discuss the current implementation status and other language features. In Section 13 we briefly further discuss related languages. Section 14 concludes.

2 Declarative and imperative programming

Declarative programming is about manipulating *values*, independently of how they are represented, stored etc and variables are just names for values. In contrast, imperative programming is about storing values in memory locations and variables are primarily names for memory locations. Memory locations have two

[1] We currently support a temporary syntax chosen to avoid us having to write a parser

distinct values associated with them: the address and the contents of the address (an l-value and an r-value). A key difference between declarative and imperative programming is the ability to perform destructive update using l-values. Destructive update of atomic values, though superficially problematic for the declarative view, does not actually cause a significant rift between the declarative and imperative paradigms. For example, consider the C code below. The occurrence of `x` on the right side of the last equation can be viewed as the value 43, whereas the occurrences on the right sides of the previous two equations can be viewed as the value 42 and the occurrences on the left side of equations can be viewed as an l-value or memory address (the same as `&x` in other contexts). Thus there are three very different meanings for the same symbol.

```
int x,y,z;       x = 42; y = x;   x = x+1;  z = x;
```

However, the variable can be renamed statically as follows:

```
int x0,x1,y,z; x0 = 42; y = x0; x1 = x0+1; z = x1; ...
```

This single assignment form can be viewed in a declarative way — replacing the variable declaration by "`let`" results in valid Haskell code. Similarly, loops can be replaced by recursive functions and other transformations can be performed to convert imperative style code into declarative code in a reasonably simple way.

A deeper difference between declarative and imperative programming is the destructive update of non-atomic values, represented using multiple memory locations which may be updated independently. When a non-atomic variable is updated, other variables may be updated as well, due to sharing of representations. For example, in the C code below which uses linked lists, the call `update(x)` may update `x` (even though C uses call by value) and also update `y`. Updating shared structures typically cannot be easily expressed in a declarative way. A syntactically simple assignment such as `*p=42` in C can be far more complex semantically because it may be updating one or more components of several variables. Declarative languages typically must expose such complexity by using more complex source code which is also less efficient.

```
list x,y; x = init(); y = x; update(x);
```

Updating shared structures is vital to many important efficient algorithms. Unfortunately, despite the many advantages of declarative programming languages, they tend to be rather clumsy and inefficient at best for implementing these algorithms. For example, consider the problem of converting an expression into a normal form (the essence of executing a functional program). In a simple case we may have a numeric expression which can be the product of two expressions, the sum of two expressions, etc.

```
data Expr = Times Expr Expr | Plus Expr Expr | ...
```

When evaluating an expression such as `Times zero x`, ideally we would like to avoid evaluating `x` (which could be a complex sub-expression) — outermost (eg, lazy) evaluation can avoid unnecessary evaluation. However, when evaluating `Times ten x` it is also important to avoid re-evaluating `x`. Rather than rewriting the expression to `Plus x (Plus x (...))` with ten occurrences of `x` which are evaluated separately, we want a single occurrence of `x` and multiple references or pointers to it. There is then a single evaluation of `x`, which is destructively updated, and ten dereference operations (which take constant time). Many functional programming languages use lazy evaluation but they are not convenient languages for implementing lazy evaluation.

Similarly, unification is at the core of most logic programming languages. We may have a variable `X` bound to variable `Y` which is bound to `Z`. When `Z` becomes bound to some other term, both `X` and `Y` also become bound to it. In the Warren Abstract Machine [1], `X` would be a (tagged) pointer to `Y`, which would be a pointer to `Z`, which would point to itself. Binding `Z` simply updates the word containing `Z`. Similarly, the Parma [2] implementation scheme uses cyclic lists of pointers to represent variables. Although there are declarative languages where an algebraic data type such as the one below results in the desired representation for logic variables, they are not convenient for expressing details of an efficient unification algorithm.

```
data Term =
    Var Term | -- variables are pointers (possibly to self)
    Nonvar Constructor [Term] -- could refine this
```

One of the main aims of Pawns is to have a language with high level features like typical declarative languages, but which also allows low level programming involving pointers and update of shared data structures. The challenge (in common with many similar attempts in the past) is to allow the latter without compromising the former. Typically, if a function has some "effect", such as updating a shared data structure, it cannot be simply viewed as a function in the mathematical sense, and neither can any function which (directly or indirectly) calls it. Thus a single destructive update in a single function can destroy the declarative view of the whole program. Eliminating impurity completely can make it difficult to implement certain algorithms efficiently, leading to frustration for programmers and cries such as "I love purity, but it's killing me"[2]. Various languages have been designed to allow impurity to be encapsulated so as to limit its impact on declarative code. Pawns goes further than most of these languages in allowing low level code (it is closer to C in that respect) and uses some novel techniques for encapsulation.

3 Three views of algebraic data types

We now present three different views of algebraic data types. We start with the "high level" view, the typical view taken in most functional languages, then progressively consider lower level views. Consider the following definitions of two

[2] `http://www.haskell.org/pipermail/haskell-cafe/2008-February/039339.html`

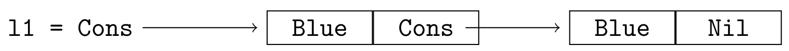
data types and a value. The type `Colour` has three possible values, all of which are atomic. The type `Colours` (lists of colours) is recursive and uses both a sum (a `Colours` is a `Nil` or a `Cons`) and a product (for a `Cons` we have have both a `Colour` and a `Colours`). The value `l1` is a list containing two occurrences of `Blue`. In the high level view we typically consider the values of variables but not where they are stored.

```
data Colour = Red | Green | Blue
data Colours = Nil | Cons Colour Colours
l1 = Cons Blue (Cons Blue Nil)
```

In what we call the "ref" or "pointer" view, we take high level data type definitions but wrap each argument of each data constructor with a `Ref` data constructor. We also introduce a `Ref` type (the identifier `Ref` is thus overloaded, in the style commonly adopted in Haskell). This could be considered a special case of high level algebraic data types — there is no change to the type system, just a convention. However, we prefer to think of the `Ref` data constructor as a special "internal" symbol, with high level definitions being a shorthand for the ref view. High level definitions can trivially be converted to the ref form, and code using high level definitions can also be transformed in a straightforward way. Thus high level type definitions can be though of as having *implicit* refs. This is central to the design of Pawns. The ref equivalent of the definitions above follows: the colours and `Nil` are unchanged as they have no arguments, but the arguments of `Cons` have a `Ref` data constructor.

```
data Colour = Red | Green | Blue
data Colours = Nil | Cons (Ref Colour) (Ref Colours)
data Ref t = Ref t
l1 = Cons (Ref Blue) (Ref (Cons (Ref Blue) (Ref Nil)))
```

The reason for introducing the ref view is that it exposes more of the low level implementation details of the types. Specifically, it reflects the fact that for each `Cons` there are two references or addresses on the heap which contain the head and tail of the list, respectively. Data constructors with no arguments are generally represented as small integers (like an `enum` type in C). Those with arguments are represented as a tag, to indicate the data constructor, plus pointer to a block of words on the heap containing representations of the arguments. If there are a small number of possible data constructors with arguments the tag can often be incorporated into the pointer, otherwise it can be stored with the arguments on the heap. The ref view exposes how arguments are stored but no details of the tag or other information which may be stored (such as overheads associated with memory management). In the case of lists in a strict strongly typed language, `Nil` can be represented as zero (or `NULL`) and `Cons` can be represented as any other pointer value — the tag is empty. The memory layout for `l1` can be pictured as below, where the boxes represent words on the heap and `Cons` followed by an arrow represents a pointer (and empty tag). Note that `l1` itself may not be stored on the heap — it may be in a register, for example.



Note also that a value such as `Ref Nil` simply corresponds to a value `Nil` which happens to be stored on the heap. This means it has an address which can be computed, but that address is not necessarily stored anywhere. The data structure does not contain an explicit pointer to a word containing the representation of `Nil`. Similarly, the representations of first argument of each `Cons` cell, `Ref Blue`, are not pointers to a colour — they are words on the heap containing the colour itself (most likely represented as a small integer). If arguments of `Cons` were wrapped in an extra data constructor in the high level type definitions (for example, the `STRef` data constructor available in a Haskell library which allows destructive update) we would have a different data structure, with additional levels of indirection (we discuss this further in Section 4). Thus considering the high level view as a shorthand for the ref view, with the `Ref` data constructor being special, is significantly different to writing high level types containing explicit refs.

The lowest level view of the data is what we call the "store" view, where each ref is represented by a heap address (an integer), and there is a separate store (typically implicit) which maps addresses to values. This could be viewed as follows (though the simple view of a single store which maps addresses to values of multiple types does not fit easily with static type checking).

```
data Colours = Nil | Cons (Ref Colour) (Ref Colours)
data Ref t = Ref Int
l1 = Cons (Ref 65536) (Ref 65544)
l1t = Cons (Ref 70000) (Ref 70008)
store = {65536->Blue, 65544->l1t, 70000->Blue, 70008->Nil}
```

To determine a (high level) value we need the value of the store and if a single value in the store is changed, several high level values may change. For example, if the store changed so it maps 70000 to `Red`, both the high level values `l1` and `l1t` would effectively change. This view is needed to understand destructive update. Neither the high level view or the ref view expose sharing of data structures. The store view does expose sharing and many declarative programmers routinely use this view (or an approximation to it) when considering the efficiency of their code. Generations of students have been taught a little about the implementation of declarative languages so they understand, for example, that adding an extra element to the front of a list is a $O(1)$ operation (the tail of the new list is shared) whereas adding an extra element to the end of a list is $O(N)$, where N is the length of the list. Although low level views are not required for correctness of code, they can be important for efficiency. Expert programmers can experience a degree of frustration when they understand the store view of their data structures but have no way of destructively updating them. Pawns is an attempt to remedy this without sacrificing too much.

4 (De)constructing and updating values in Pawns

We now describe how values are deconstructed, constructed and updated in Pawns. This is the key to how Pawns combines the high level view of data and the low level views. Later we describe additional annotations etc which must be added to avoid compiler errors and which make it clear when code must be viewed at a low level.

Consider the high level method of replacing the head of a list `cs0` with `Red`. We use some form of pattern matching to deconstruct the list into a head and a tail, then construct a new list with the same tail but `Red` as the head:

```
case cs0 of (Cons c cs) -> Cons Red cs
```

In Pawns we can expose the ref or pointer view by putting “`*`” before variables. Below we also rename the variables. Just as `cs` above denotes a list (the type being inferred by the compiler), the expression `*csp` below denotes a list. The variable `csp` alone is a *pointer* to a list. The syntax is inspired by C, where we can have declarations such as `list cs` and `list *csp`. The expression `*csp` denotes the list pointed to by `csp`; in a pattern it binds `csp` to a pointer to the matched list.

```
case cs0 of (Cons *cp *csp) -> Cons Red *csp
```

Note that this code does exactly the same deconstruction and construction as the high level version. The main difference is that the programmer can view it in a lower level way, thinking of pointers to values which are stored in memory locations rather than just values which exist at a more abstract level. The inferred type of `csp` is `Ref Colours`. Taking the ref view, normal patterns (such as the high level one above containing `cs`) contain implicit `Ref` data constructors. A “`*`” prefixing a pattern argument means the `Ref` is omitted. Thus `csp` may be bound to a value such as `Ref Nil`, which can be represented simply by a pointer to a heap address containing `Nil`. We call patterns and expressions containing “`*`” *dereference patterns* and *dereference expressions*, respectively. They do not introduce impurity into the language (there is just a slightly different rule for type inference with dereference patterns and expressions), but do provide a lower level view of data types which is useful for support of impure operations such as destructive update.

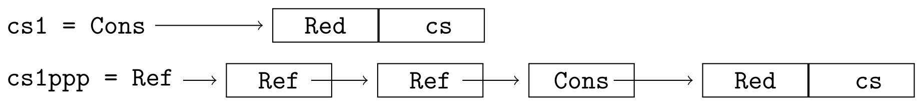
Destructive update is done via pointers in Pawns, also in a similar way to C. This is the only way destructive update is done in Pawns. Note that the code below will not actually be accepted by the compiler without some additional annotations (see Section 5). Using the same pattern matching as above we obtain a pointer to the head of the list, which can then be destructively updated with the value `Red`, followed by returning the (now modified) list `cs0`. In Pawns an expression/statement can be a sequence of sub-expressions/statements which are evaluated in turn, with the value of the last sub-expression being returned as the value of the whole expression (“`;`” in Pawns is like `‘seq‘` in Haskell).

```
case cs0 of (Cons *cp *csp) -> *cp := Red; cs0
```

As well as obtaining pointers using dereference patterns, Pawns allows pointers to be created in one other way, using an extension of “let” bindings with “*” occurring on the left. This adds no significant expressive power but is convenient and also allows pointers to pointers, etc. The first equation below is essentially a standard let binding of the variable `cs1`. It can be implemented by allocating space for a cons cell on the heap, initializing it appropriately and storing an appropriately tagged pointer to it in `cs1`. The second equation creates the same list. However, it binds the variable `cs1ppp`, which is a pointer to a pointer to a pointer to a list, to the value `Ref (Ref (Ref (Cons (Ref Red) (Ref cs))))` using the ref view (the last two refs are implicit in the high level view). This is most simply implemented by allocating additional space on the heap for three pointers (the last of which has a `Cons` tag associated with it) and initializing them appropriately.

```
cs1 = Cons Red cs;     ***cs1ppp = Cons Red cs
```

The memory layout can be pictured as follows, where `Ref` (and `Cons`) followed by an arrow is a raw pointer:



This completes the picture of how Pawns incorporates destructive update into a declarative language. Dereference patterns allow us to obtain refs/pointers to any argument of any data constructor (in the high level view) and use a single primitive to destructively update them. Note however, there is no explicit `Ref` data constructor or primitive in the language, and no equivalent of “&” in C. Only dereferencing is explicit, using “*”. The fact that references are implicit is important because the place where they occur most frequently is in high level code. Making references explicit in the high level code above would lead to the following clutter, greatly reducing clarity and abstraction:

```
case cs0 of (Cons (Ref c) (Ref cs)) -> Cons (Ref Red) (Ref cs)
```

In the lower level code, occurrences of “*” essentially cancel out the references (they are inverse operations). Because references are implicit in Pawns, we can write and view the high level code in the same way as a typical “pure” declarative code. We can also view all code at a lower level as well, but this is only necessary when the code does low level things, such as destructive update.

Implicit references allow us to use a single data type definition, such as `Colours`, but update it in flexible ways. In other functional languages which support (mutable) references such as refs in ML [3] and STRefs in Haskell, there can be four different versions of this data type depending on which subset of data constructor arguments we want to update (a type with N data constructor arguments has 2^N variants). This can lead to overheads due to conversion from one variant to another and additional code for performing conversions and/or duplicating code to handle more than one variant (there are 256 variants of the Haskell Prelude function `zip`, for example). Converting Haskell code which uses a non-mutable version of a data type to an mutable version is particularly cumbersome because the latter must use monads.

Implicit references also allow us to update any data constructor argument without compromising the efficient representation of values. Where references are explicitly included in user-defined types the representation of the type must include an extra level of indirection. This is not simply a matter of compiler optimisation, since it is exposed in the semantics. For example, lists which can be mutable must have refs, and with explicit refs it is possible to have two distinct list elements which are identical refs. Taking the "ref" view of our type `Colours` (or the equivalent in ML or Haskell), we can have code like the following:

```
l1 = let rb = Ref Blue in Cons rb (Ref (Cons rb (Ref Nil)))
```

If the head of this list is updated to be `Red`, the second element is also updated, exposing the fact that the refs are identical and requiring the implementation to use pointers to colours in the representation. Such behaviour cannot occur with type `Colours` in Pawns code because `Ref`s are implicit. We discuss this further in Section 13. One final advantage of the support for pointers in Pawns is that it prevents code like the C code below, which returns a pointer to a local variable allocated in a stack frame which is popped. In Pawns, a local integer variable can safely be represented by an unboxed integer on the stack or in a register.

```
int *oops(){int i = 42; return &i;}
```

The potential disadvantage of having C-like ability to destructively update data structures is that we no longer have a declarative language. We need to "tame the beast" of destructive update in order to get the advantages of declarative coding. The main mechanism for this is described next.

5 Destructive updates are obvious in Pawns

As mentioned in the previous section, the following code results in a compiler error in Pawns:

```
case cs0 of (Cons *cp *csp) -> *cp := Red; cs0
```

The reason is that whenever a variable may be modified in Pawns, the code must make it obvious by annotating the variable with "`!`"[3]. The value of `cp` may be modified by the assignment (the pointer itself is not modified but what it points to may be modified, and this is considered part of the value when viewed at a high level), so `cp` must be prefixed with "`!`". Furthermore, `cp` points to part of `cs0`, so the assignment may also modify `cs0`. Since `cs0` is used subsequently, this must be declared as well with an additional "`!cs0`" annotation attached to the whole statement (hence the extra parentheses):

[3] The only exception is variables which are no longer "live" — they are not used in subsequent code and are not arguments of the function.

```
case cs0 of (Cons *cp *csp) -> (*!cp := Red) !cs0; cs0
```

The name Pawns is an acronym for

Pointer Assignment With No Surprises

Pawns code which has no occurrence of “!” has no observable effects, behaves in a purely declarative way and can be viewed in a high level way. Effects such as destructive update of shared structures are possible in Pawns but the extent of any effects are always made clear in the source code. In contrast, imperative languages allow effects which are far from obvious. This is a significant disadvantage, since understanding and controlling code inter-dependencies is a major challenge of programming.

In order to detect all variables which may be modified by an assignment, the Pawns implementation does *sharing analysis* to determine which (parts of) different variables may share or alias. This analysis can be imprecise and the implementation may conclude there is possible sharing between two variables when in fact there cannot be any sharing. The consequence of this is that additional variable annotations may sometimes be needed to prevent compiler error messages. Pawns essentially places no restrictions on what arguments of data constructors can be updated and where, as long as the appropriate annotations etc are added to the program. In the worst case the compiler may conclude that every variable in a function definition may share with every other variable, and thus every variable must be annotated for each assignment. We hope that most Pawns code will use the pure functional style, not requiring any annotations, and even for impure code the worst case scenario will occur very rarely.

To help analyze update and sharing involving function calls, there are two other features of Pawns. The first is that type signatures can have annotations to indicate some arguments are mutable (may be updated) and/or data structures should be viewed at a low level. The second is that functions can have preconditions and postconditions declared which describe what sharing is permitted at the time the function is called and what additional sharing may be present when the function returns. We will describe these features using examples in the following section. Like type information, mutability and sharing information can be inferred. However, although the compiler may provide support for inference (such as error messages suggesting appropriate declarations to include), mutability and sharing must be considered by programmers, and should be documented, so we believe it is best for the source code to include the declarations.

6 Examples

We now present some simple examples of Pawns programs which illustrate destructive update and other features.

6.1 Building a binary search tree from a list

Consider building a binary search tree from a list in the imperative style, which starts with an empty tree and repeatedly destructively updates it to insert new items. We assume there is a data type called `Item` which supports comparison. The types `List`[4] and `Tree` are defined below, along with the top level function which converts a `List` to a `Tree`. Note that the type signature of `list_bst` contains no extra annotations. Although `list_bst` is defined in terms of impure functions which have effects, these are encapsulated — any caller of `list_bst` can consider it to be a pure function without effects (see Section 8). Similarly, even though the tree is updated inside the function, the `Tree` type is identical to what would be used in a pure function.

```
data List = Nil | Cons Item List
data Tree = Empty | Node Tree Item Tree

list_bst :: List -> Tree
list_bst xs =
    *tp = Empty           -- create pointer to empty tree
    list_bst_du xs !tp    -- insert items into tree, updating it
    *tp                   -- return tree
```

Initially an empty tree is created, and a pointer to it is maintained and passed around. The call to `list_bst_du` has two arguments: the list and the pointer to the tree. The latter is annotated with “`!`”, meaning its value can be modified by the function call. The type signature of `list_bst_du` also indicates that this argument may be modified: `Ref !Tree` indicates that the tree may be modified, which implicitly also modifies the reference[5]. This function returns `()`; the effect of updating the tree is its only contribution to the computation. It simply recurses down the list, inserting each single element into the tree using the function `bst_insert_du`. The `tp` argument must be annotated in this and the recursive call.

```
list_bst_du :: List -> Ref !Tree -> ()
list_bst_du xs !tp =
    case xs of
    (Cons x xs1) ->
       bst_insert_du x !tp  -- insert head of list into tree
       list_bst_du xs1 !tp  -- insert rest of list into tree
    Nil -> ()
```

The type signature of `bst_insert_du` also indicates its second argument can be updated, and it returns `()`. If the (pointed to) tree is empty, it is updated with a node containing the new data item and two empty subtrees. Otherwise, pattern matching is used to extract the item in the root node and pointers to the left and right subtrees. After comparing the new item with the item in the root, the item

[4] We can’t use `Cons` and `Nil` for both `List` and `Colours` in the same program. In practice the polymorphic list type would be used but we defer discussion of polymorphism until later.

[5] Currently we only support a somewhat lower level way of attaching annotations such as this to type signatures, described later.

is recursively inserted into the appropriate subtree. The variable representing the pointer to the subtree may be updated by the recursive call, hence it must be annotated. Furthermore, `tp` must be annotated at both recursive calls because the each subtree is part of it.

```
bst_insert_du :: Item -> Ref !Tree -> ()
bst_insert_du x !tp =
    case *tp of
    Empty ->
        *!tp := Node Empty x Empty  -- insert new node
    (Node *lp n *rp) ->
        if x <= n then
            (bst_insert_du x !lp) !tp
        else
            (bst_insert_du x !rp) !tp
```

Our Pawns code is (arguably) more elegant than typical imperative code (C, for example) which maintains a pointer to a node and its parent, and has special cases to deal with empty trees and when the traversal reaches a `NULL` pointer (often the last comparison has to be repeated to find whether the `NULL` was a left or right child). This is primarily due to our use of a pointer to a tree, which corresponds to a pointer to a pointer to a struct in C. C programmers tend to avoid pointers to pointers, leading to less elegant code. Algebraic data types can hide one level of indirection and describe data in more high level concise and precise way than C.

The Pawns code is also more efficient (and slightly shorter) than typical declarative code, which after deconstructing the tree into a node with an item and subtrees, explicitly reconstructs a new tree with a new version of one of the subtrees (see below). It is difficult for a compiler of a declarative language to optimise the declarative code so the node can be reused. Although the tree is single-threaded through the code of `list_bst` and `list_bst_du`, it is not single-threaded in `bst_insert_du`: the recursive calls have a subtree passed to them while there is still a live reference to the whole tree (a related limitation is noted in [4]). Indeed, `bst_insert_du` does not always compute the same thing as its declarative counterpart and rather subtle analysis of all three functions is required to show that a destructive operation can be used.

```
bst_insert_pure :: Int -> Tree -> Tree
bst_insert_pure x t0 =
    case t0 of
    Empty ->
        Node Empty x Empty
    (Node l n r) ->
        if x <= n then
            Node (bst_insert_pure x l) n r
        else
            Node l n (bst_insert_pure x r)
```

6.2 A cord data type

Cords are data types which support similar operations to lists, but concatenation can be done 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.

```
data Cord = Leaf List | Branch Cord Cord
```

To convert such a cord to a list, a declarative program would typically copy each cons cell in each list. An imperative 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. For each list in the tree other than the rightmost one, the `NULL` pointer at the end of the list is replaced with a pointer to the first cell of the next list; the first list is then returned. This algorithm can be coded in Pawns. The top level function creates a pointer to an empty list and calls an auxiliary function which traverses the cord, updating the cord and the pointed to list, which is then returned.

```
cord_list :: !Cord -> List
cord_list !xc =
    *xsp = Nil
    cord_list_a !xc !xsp
    *xsp
```

The auxiliary function, `cord_list_a` takes a cord and a pointer to the `Nil` at the end of the previous list (or newly created empty list for the top level call), both of which are modified. It returns a pointer to the `Nil` at the end of the last list in the cord. The “?” annotation in the type of the return value indicates that a low level view of this value should be used — its sharing is important and it may be updated subsequently (we discuss this further in Section 8). Thus if the cord is a leaf, its contents is used to update the previous `Nil` (pointed to by `np`) and we return a pointer to the `Nil` at the end of this list (this is found by the function `lastp`). If the cord has two subtrees, the first one is processed recursively, updating `np`, and the returned pointer to `Nil`, `np1`, is used in the recursive processing of the second subtree, the result of which is returned.

```
cord_list_a :: !Cord -> Ref !List -> Ref ?List
-- np points to Nil of list we smash; returns ptr to new Nil
cord_list_a !xc !np =
    case xc of
    (Leaf xs) ->
        (*!np := xs) !xc!xs    -- xc, xs are safe
        lastp np               -- return ptr to Nil of updated np
    (Branch xc1 xc2) ->
        (np1 = cord_list_a !xc1 !np) !xc!xc2 -- xc2 is safe
        (cord_list_a !xc2 !np1) !xc!np
```

The lastp function takes a pointer to a list, traverses the list and returns a pointer to the Nil at the end. Sharing of the argument and the result is important, so the low level view of the list is needed.

```
lastp :: Ref ?List -> Ref ?List
lastp xsp =
    case *xsp of
    Nil -> xsp
    (Cons _ *xsp1) -> lastp xsp1
```

In the last recursive call to cord_list_a, clearly the subtree and pointer to Nil (np1) may be modified and thus must be annotated. In addition, xc (the whole cord) is modified and so is np (since *np1 is a suffix of *np), so these variables must be annotated. Similarly xc must be annotated in the first recursive call. In addition (and perhaps surprisingly for a programmer), the compiler insists that the second subtree is annotated at this point and both the tree and the list in the leaf are annotated when a leaf is encountered and np is updated. The code here includes comments suggesting that these annotations are "false positives" of the sharing analysis. As well as these annotations on variables, the programmer must consider what sharing may be possible when cord_list_a is called, and when it returns. These three aspects of sharing are inter-dependent. For example, more sharing in a top level call may mean that more variables must be annotated, there is more sharing in recursive calls and more sharing when the result is returned.

Type signatures on functions can have annotations which declare preconditions and postconditions concerning sharing, as well as mutability. Preconditions describe the maximum possible sharing whenever the function is called and postconditions describe the additional sharing introduced by calling the function, so the union of the two describes the sharing when a function returns. The most straightforward declarations accepted by the compiler are shown below[6]. The syntax for pre- and post-conditions currently supported by pawns is the same as statements in function definitions. It describes the sharing which is possible if (some of) the code is executed or (some of) the equations are true. For example, in the postcondition of cord_list_a, np1 may equal np, so it is assumed all their components may share. In addition, if xc is a leaf, its content may share with the list pointed to by np, and therefore also np1. The sharing analysis doesn't distinguish between the overall sharing of a cord and the sharing of its subtrees, so the sharing above is assumed to hold for all leaves of xc. The sharing between the leaves and *np must be in the precondition, rather than just the postcondition, because of the double recursion in the function definition. The keyword "nosharing" can be used to indicate there is no sharing.

[6] The location of the "!" annotation is different to the higher level syntax we have shown and explicit sharing is declared rather than "?" annotations. We plan to support the previous syntax, with the default being no sharing for annotated arguments.

```
cord_list_a :: Cord -> Ref List -> Ref List
  sharing cord_list_a !xc !np = np1
    pre xc = Leaf *np
    post np1 = np; xc = Leaf *np

lastp :: Ref List -> Ref List
  sharing lastp xsp = xsp1
    pre nosharing
    post xsp1 = xsp
```

It may seem that the compiler is being overly conservative in insisting on some of the annotations, and some of the possible sharing is detects is bogus. However, the possible sharing detected *can* occur if there is sharing between the different lists in the cord. Indeed, if that occurs, the code will not work correctly — it can create a cyclic list then loop. Performing the analysis and insisting on the programmer providing annotations and declarations makes it significantly more likely the programmer will realise this potential bug. Compared with pure declarative code, the code is indeed “dangerous” and it is appropriate that the compiler helps the programmer understand this fact. Similarly, anyone reading the code is more likely to be aware of the subtleties (and one would hope that the programmer, having understood the subtleties, would also document them in comments).

Cords containing shared lists can be avoided by additional use of preconditions and a restricted interface to the cord data type. Pawns allows us to build an interface to cords which maintains the safety condition. For example, the interface could include a function which converts a list to a cord, by simply applying the `Leaf` data constructor, and a function which appends a list onto a cord, by using `Leaf` and `Branch`. The latter can have a precondition that the list does not share with the contents of any leaf of the cord. Any call to this function which violates the precondition would then result in a compiler error:

```
list_cord :: List -> Cord
  sharing list_cord xs = xc
    pre nosharing
    post xc = Leaf xs
list_cord xs = Leaf xs  -- just wrap the list in a Leaf

cord_app_list :: Cord -> List -> Cord
  sharing cord_app_list xc xs = xc1
    pre nosharing  -- NOTE: xs must not share with lists in xc
    post xc1 = Branch xc (Leaf xs)
cord_app_list xc = Branch xc (Leaf xs)

...
    xc0 = list_cord xs         -- create cord xc0 from list xs
    xc1 = cord_app_list xc0 xs -- ERROR: xc0 and xs share
```

7 Overview of sharing and mutability analysis

We now give a very brief overview of the (rather complicated) algorithms used to check that sharing and mutability declarations are consistent with the code. The sharing analysis for Pawns is similar to that done in other declarative languages such as Prolog [5], Mercury [6] and Mars [7] but has additional support for pre- and post-conditions, destructive update and pointers. For each function, the code is abstractly interpreted to compute a conservative approximation to sharing at each program point. The sharing information is initialised with the precondition of the function. The sharing computed for when the function returns must be a subset of that in the union of the pre- and post-condition. Where a function is called, the sharing information, projected onto the function arguments, must be a subset of the precondition declared for the function. The program point just after the function call has additional sharing information. It is the union of its pre- and post-conditions, projected onto the function result and the arguments which are declared mutable in the type signature of the function.

Sharing analysis of let (equality) and case statements is very similar to that done in other languages. For case statements, each branch of the case is analysed separately and the overall result is the union of the sharing information from each branch. For each branch, analysis typically eliminates some sharing information. For example, in a `[]` branch we eliminate all sharing for the list variable. Let bindings introduce new sharing information between (components of) the bound variable and the variables on the right hand side. Consider the following example using the Haskell type `[Maybe Int]`:

```
xs = (Just i):y1:y2:ys
```

Lists can be arbitrarily long but “type folding” merges information from the tail of the list with the whole list, leading to just three distinct components of `xs` for sharing analysis: the “spine” and elements of the list (the two arguments of cons) and the argument of `Just` within the elements. The abstract domain we use for sharing is a set of pairs of variable components which can be updated. After this let binding, the spine and element components of `xs` share with the spine and element components of `ys` and the “Just” component of `xs` shares with the “Just” components of `ys`, `y1` and `y2`. Any variables which previously shared with `y1`, `y2` or `ys` also share with the corresponding components of `xs`.

Note that there is no sharing between `y1` and `y2` introduced. Also, the “Just” components of `y1` and `y2` are their only components for sharing analysis and `i` has no components since it is an (unboxed) integer (components correspond to `Ref` data constructors in the “ref” view of the type). The abstract domain also contains information about what components of a variable may exist. For example, if it is known that `y1=Nothing` (from a previous binding or a case expression) then `y1` will have no `Just` component so no sharing between `y1` and `xs` will be introduced.

Analysis of an assignment to `*v` is similar to that for a let binding in that the same sharing information is added. However, additional sharing may also need to be added for variables which have a component that aliases `*v`. For example, we may know that *v and *w may be aliases and *w=[]. There are no shared list elements of *v and *w but an assignment such as *!v:=[e] would introduce such sharing (this assignment must be annotated with !w). Also, existing sharing for components of *v can be dropped in some cases to improve precision. For example, if e shares with just the elements of the list *v, this sharing can be dropped after the assignment *!v:=x, assuming x does not share with v or e.

Mutability analysis uses sharing analysis. Assignment statements must have a "!" annotation for the variable on the left and additional annotations for each live variable which has a component which may alias it. Similarly, variables in arguments of function calls must have a "!" annotation if the type signature of the function declares that argument position mutable and there must be additional annotations for live variables which may share with those annotated variables. Finally, any formal parameter of a function which has a "!" annotation within the function definition must be declared mutable in the type signature.

8 Mixing high and low level code

So far we have given examples of low level Pawns code which performs destructive update. Pawns also supports more traditional high level functional programming using the same types — the Tree type can be used in the same way as it would be in ML or Haskell. In addition, the type signature of list_bst indicates it does not mutate its argument or require a low level view. Below we give a (not so good) way of computing the length of a list and the intention is that reasoning about correctness of code such as this should be the same as in a pure functional language, using the high level view of the types and without the need to consider how values are stored and what sharing exists.

```
len :: List -> Int  -- returns the length of a List
len l = bst_size (list_bst l)

bst_size :: Tree -> Int  -- returns the size of a Tree
bst_size t = case t of
    Empty -> 0
    (Node l _ r) -> (bst_size l) + (bst_size r) + 1
```

A function with a "!" annotation in the type signature indicates the function is not pure — it may have effects (mutating such annotated arguments) and any reasoning about it must use the low level "store" view for annotated arguments, including information about sharing. A function with just "?" annotations or explicit sharing in its type signature, such as lastp, behaves as a pure function since it has no effects, but the high level view does not give the whole story. It neglects sharing information which is typically necessary for verification of the program as a whole. Functions with no annotations on their type signatures, such as list_bst, are pure and the high level view should be sufficient for verification. Two important considerations are the nature of the lower level reasoning and how high and low level code can be mixed, both from the perspective of verification and sharing analysis.

It is reasonably straightforward to transform Pawns code to make the store view more explicit. Our prototype implementation has code to do this in the case there is a single type in the store — transforming Pawns into Haskell so as to make use of the store unambiguous and using Haskell in an imperative style [8]. Types are transformed so they use `STRef` wrappers around arguments, to enable destructive update, and nested expressions are "flattened" into a sequence of operations using monadic "do" notation with `readSTRef` and `writeSTRef` primitives to access and update the data structures. A Haskell version of `bst_size` using the modified type is given below. When multiple types are required in the store, translation into Haskell is more complex as we need a stack of `ST` monads. Reasoning about partial correctness of the Pawns code can be done in essentially the same way as reasoning about the monadic Haskell code which exposes the additional complexity of the store view. We do not discuss the details of reasoning about high or low level code here, but separation logic [9] provides a useful basis for formal reasoning about code which updates shared structures.

```
data DUTree s =  -- store view of Tree for destructive update
    DUEmpty |
    DUNode (STRef s (DUTree s)) -- mutable reference to subtree
        (STRef s Item) (STRef s (DUTree s))

dubst_size :: DUTree s -> ST s Int  -- size of a DUTree
dubst_size t =
    case t of
    DUEmpty ->
        return 0
    (DUNode lp _ rp) ->   -- node with refs/pointers to subtrees
        do
        l <- readSTRef lp     -- dereference lp using store
        sl <- dubst_size l    -- get size of l using store
        r <- readSTRef rp     -- dereference rp using store
        sr <- dubst_size r    -- get size of r using store
        return (sl + sr + 1)  -- return total size
```

It is possible to take the store view for all Pawns code, or transform it all to Haskell. However, this "throws the baby out with the bath water" since it significantly increases the complexity of pure code such as `bst_size`. What seems preferable when reasoning about correctness is to use the store view in some places (such as `list_bst_du`) and the high level view in others (such as `bst_size`), and convert from the former to the latter where needed. This is similar to verifying correctness of a program which converts from an mutable tree type such as `DUTree` to a high level version of the data type such as `Tree` which is not mutable (this can be done in an unambiguous way). In Pawns the conversion can be done in the correctness proof so there is no need to actually write, verify or execute code which converts from a `DUTree` to a `Tree`.

Using two distinct views of values can thus greatly simplify reasoning about Pawns code and the annotations in type signatures help document which view can be used where, and maintain consistency. For Pawns functions with unannotated type signatures, parameters and results are considered "abstract". For annotated type signatures, annotated parameters and results are considered "concrete", as are local (let-bound) variables by default. Concrete variables and the store view can be used as we build and update data structures using data constructors and functions with annotated type signatures. Concrete variables can also be viewed in an abstract way. We can simply ignore the concrete store view and consider (or convert to) the abstract high level value it represents when values are passed to or returned from a function.

Note however that conversion in the other direction, from abstract to concrete, cannot be done unambiguously in general, so an abstract value should not be used where a concrete value is required and sharing information is specified. Similarly, concrete values can contain abstract values but not the reverse. For example, a concrete tree can contain abstract values in the nodes but an abstract tree can only contain abstract values in nodes. Sharing of subtrees implies sharing of values within them, so if the sharing in the tree is unknown the sharing in the nodes must also be unknown.

The sharing analysis of Pawns has extra features to support abstract variables and mixing of high and low level code. Declaring several variables abstract is like saying they can all share with each other and, more importantly, they also share with a special variable named `abstract` which is passed to every function. More precisely, all abstract variables of the same type may share and there is a separate variable $\texttt{abstract}_t$ for each type t, since variables of different types cannot share. A key restriction is that no variable which shares with `abstract` can be updated, even if it is annotated with "!"[7]. This is enforced when the compiler checks "!" annotations. Sharing with `abstract` is the way the compiler distinguishes variables for which only the high level view is appropriate. Converting from concrete to abstract is like adding possible sharing with `abstract` in the sharing analysis. This sharing cannot be removed while the variable is live.

For unannotated type signatures, arguments implicitly share with `abstract` in the precondition and both arguments and results share with `abstract` in the postcondition. This can also be made explicit in the program source, allowing functions which operate on a mixture of concrete and abstract arguments. If the `lastp` function of Section 6.2 had the unannotated type signature `lastp :: Ref List -> Ref List` it would be equivalent to the following:

```
lastp :: Ref List -> Ref List
    sharing lastp(xsp) = np
    pre xsp = abstract
    post xsp = abstract; np = abstract
```

The compiler renames occurrences of "`abstract`" depending on the inferred type. Since `xsp` and `np` have the same type, `np` shares with both (the `Ref List`

[7] Again, the only exception is variables which are no longer "live".

version of) `abstract` and `xsp` in the postcondition. If `lastp` was only used in pure code this type signature would be acceptable (and we could change the function definition so it simply returns a pointer to a freshly allocated occurrence of `Nil` without any list traversal).

However, in our cord code it is vital that `lastp` returns a pointer to the `Nil` at the end of its argument. The argument of `lastp` can be returned by `lastp`, which can be returned by `cord_list_a`, which can be passed to the second argument of (a recursive call to) `cord_list_a`, which can be updated. Thus all of these arguments and results must be concrete. If the postcondition of `lastp` includes sharing with `abstract`, sharing analysis of `cord_list_a` would produce an error message. The call to `lastp` would result in the return value of `cord_list_a` sharing with `abstract`, which conflicts with the postcondition of `cord_list_a`. And if sharing with `abstract` was added to the postcondition of `cord_list_a` destructive update of `np1` would not be allowed.

For our tree building code of Section 6.1 the argument and result of `list_bst` are abstract. The second argument of the call to `list_bst_du`, `tp`, is constructed locally within `list_bst` and is concrete since there is no sharing with `abstract` introduced. The tree pointed to, `*tp`, is returned by `list_bst`. This conceptually requires conversion from concrete (inside the definition of `list_bst`) to abstract (where `list_bst` is called, for example in `len`). Sharing analysis mirrors this by adding sharing with `abstract` to the return value of calls to `list_bst` because of its postcondition.

Our coding for `list_bst` above is sufficient for many purposes but because the tree returned is abstract it cannot subsequently be destructively updated (for example, a pointer to it cannot be passed to `bst_insert_du`). Such flexibility can be achieved by returning a concrete value, using the alternative type signature `list_bst :: List -> ?Tree`, or explicitly stating there is no sharing. This version can still be used in high level code such as `len` because the concrete tree can always be viewed abstractly. During code development we can start with high level code where everything is abstract, then consider lower level details and make some data structures concrete so destructive update can be used, but without breaking abstract code which relies on it.

However, mixing high and low level code has some restrictions. In some cases, once a data structure is passed to abstract code it can't be updated because it could share with abstract variables which are used elsewhere. For example, in the code below we assume there is a function `bst_id :: Tree -> Tree`. The call to `bst_id` may introduce sharing between `*tp` and `t1`, which is abstract. While `t1` is live, `*tp` should not be updated because that may also update `t1` in ways dependent on the sharing introduced by `bst_id`. The rules of Pawns prevent us from updating `*tp` without adding `!t1` (since they may share) and prevent us from adding `!t1` because it shares with `abstract`. However, when `t1` is no longer live `*tp` can be updated again.

```
*tp = ...               -- initialize tree *tp
bst_insert_du x1 !tp  -- OK to update *tp
t1 = bst_id *tp         -- t1 may share with *tp and abstract
bst_insert_du x2 !tp  -- ERROR (even with !t1 added)
... t1 ...              -- last occurrence of t1
bst_insert_du x3 !tp  -- OK again since t1 is dead
...
```

There are other cases where we can update a concrete variable after passing it to a function which returns an abstract value. For example, in the code below there cannot be sharing between the result of `bst_size` and its argument because the types are different (furthermore, the result is atomic). The implicit precondition of `bst_size` introduces sharing between the tree and `abstract` in the analysis of `bst_size`. However, `bst_size` is pure and this sharing is not added after calls to `bst_size` in the analysis of the code below.

```
*tp = ...
bst_insert_du x1 !tp
s1 = bst_size *tp
bst_insert_du x2 !tp  -- OK
s2 = bst_size *tp
... s1 s2 ...
```

We now summarise how destructive update is encapsulated in Pawns. Data structures can potentially be viewed in two ways: the abstract high level view and the concrete store view which exposes how they are stored and what parts are shared. For abstract values only the high level view should be used for reasoning about correctness, and this can be done in the same way as pure code in other declarative languages. For concrete values both views are possible. Reasoning about correctness of code which constructs the data structure can be done with the concrete view, and this view is essential if destructive update is used, but the high level view can also be used if the data structure is passed to high level code. Reasoning about correctness of low level code relies on unambiguously sequencing all function calls which have effects and passing around the store. The store exposes the representation of data structures, including any sharing.

Pawns functions have (possibly implicit) pre- and post-conditions concerning sharing. The primary distinction they make is whether any information is known about the structure — whether the variable is concrete or abstract. For concrete variables the declarations must describe (a conservative approximation to) what sharing exists. Abstract variables can be seen as possibly sharing with other variables of the same type plus a special variable referred to as `abstract`. Function definitions must also declare what variables may be destructively updated at each point and the type signatures must declare what arguments may be updated. Checking that the declarations and definitions are consistent relies on sharing analysis which can be performed one function at a time.

9 IO and state variables

Like destructive update, IO does not fit easily with the declarative programming paradigm. Pawns models IO by using a value, representing the state of the world, which is conceptually passed in and returned from all computations which perform IO. Rather than explicitly using an extra argument and a tuple for results (causing verbose and hard to modify code), `io` is declared in the type signature as an implicit value which may be read (an argument of the function) and written (returned from the function). For example, the primitive function which reads a character has the following type signature:

```
get_char :: () -> Int
    implicit rw io
```

In a similar way, Pawns code can define and update other variables which are implicitly passed around; they are called state variables. Calls to functions which use state variables must be prefixed with "!" to indicate there is more going on than meets the eye. As with other uses of destructive update in Pawns, the rules for using state variables allow effects in a subcomputation to be encapsulated in a function which has a pure interface.

State variables must have a ref type (they are modified using destructive update) and are prefixed by "!" in their type signatures. For example, we can declare a state variable which is (a pointer to) an int as follows:

```
!counter :: Ref Int
```

Functions can use state variables by including the `implicit` keyword in the type signature. A state variable can be declared read-write (`rw`), read-only (`ro`) or write-only (`wo`). A state variable is introduced into a subcomputation by calling a function where it is defined `wo` (which creates a binding for the state variable) and subsequently used by functions which declare them `ro` or `rw`; the latter also allows update of the state variable. The `io` state variable is conceptually defined prior to the top level call to the function `main` in a Pawns program. Postconditions can contain any state variables declared in the type signature and preconditions can contain `rw` and `ro` state variables.

Consider the following code, which sums the elements of a binary search tree of integers. The top-level function, `bst_sum` behaves as a pure function, as indicated by the type signature. However, it is implemented by setting `*counter` to zero, calling a function `bst_sum_du` which traverses the tree and updates `*counter` at each node, then returning the final value of `*counter`.

```
bst_sum :: Bst -> Int
bst_sum t =
    !init_counter 0  -- like counter = 0
    !bst_sum_du t    -- like counter' = bst_sum_du t counter
    *counter         -- like counter'
```

```
init_counter :: Int -> ()
    implicit wo counter
init_counter n =
    *counter = n
```

Because counter is not an implicit argument of bst_sum, it is necessary to call the wo function init_counter before the counter can be used by bst_sum_du. The add_to_counter function has a similar type signature to init_counter, but the implicit argument is rw rather than wo. This means it can only be used in a context where counter is already defined, for example, in bst_sum_du, where it is defined because it is an implicit rw argument. Our coding of bst_sum_du processes the right subtree by calling bst_sum. Although this call uses counter in a sub-computation, it is a pure function so the effects are encapsulated and it does not change the value of counter in the outer computation. As we recurse down the right branch of the tree, there are effectively multiple versions of counter, just as there are mutiple versions of the arguments of a recursive function.

```
add_to_counter :: Int -> ()
    implicit rw counter
add_to_counter n =
    *!counter := *counter + n

bst_sum_du :: Bst -> ()
    implicit rw counter
bst_sum_du t =
    case t of
    Empty -> ()
    (Node l n r) ->
        !add_to_counter n    -- add n to counter
        !bst_sum_du l        -- add ints in l to counter
        -- !bst_sum_du r     -- (could code it this way)
        !add_to_counter (bst_sum r) -- add bst_sum r to counter
```

State variables are very similar to mutable global variables (and are implemented using global variables, with saving to and restoring from local variables where needed). However, the "surprises" associated with mutable global variables are avoided. State variables can only be used where they have been passed into a function as an implicit argument declared in the type signature, or after a call to a function where they are declared wo in the type signature, and all implicit uses are marked with "!". The only state variables a function can depend on or visibly affect those that are declared in the type signature of the function.

10 Polymorphism

Parametric polymorphism is a very attractive feature of many declarative programming languages. It helps support code reuse and abstraction. Adding destructive update in a naive way can mean that execution does not preserve well-typedness. Our general philosophy is that pure code which supports only the high level view should be have flexibility similar to other functional languages. There may well be somewhat annoying restrictions on impure code but, at worst, this should result in some code duplication.

The problem with type safety or "preservation" is illustrated by the following code. We construct a reference/pointer to the polymorphic value `[]` and apply two different functions to it (requiring two different instances of the type). Since the reference is mutable we can (potentially) assign another value to the variable (either directly with an assignment statement or indirectly via a call to an impure function) with an instance of the original type. If the value is now passed to a function with a conflicting type, type preservation is violated.

```
*xsp = []         -- [] has polymorphic type [t]
a = and *xsp      -- apply and :: [Bool] -> Bool - OK
s = sum *xsp      -- apply sum :: [Int] -> Int - OK
assign !xsp [3]   -- update *xsp with an instance of its type
a1 = and *xsp     -- Problem: and applied to [Int]
```

One solution to this problem is the value restriction in ML which disallows polymorphism in let-bound variables if the right side of the equation is not a "syntactic value" [10]. References are not considered syntactic values so the original binding of `xsp` above is not allowed. This would not be a practical solution in Pawns due to the implicit refs in all data structures. However, because Pawns programs contain more precise information about what can be updated, Pawns can adopt a more flexible approach. In an assignment to `*v` the type of `v` without its outer `Ref` must be an instance of (not more general than) the type of the value assigned. Similarly, if `v` is passed as a mutable argument of a function call, its type must be an instance of the inferred type of the argument. Given the type signature of `assign` is `Ref !t -> t -> ()` the inferred instance is `Ref ![Int] -> [Int] -> ()`. The attempt to destructively update `xsp` above is rejected by the compiler because it has type `Ref [t]`. This rule is not a practical solution in ML because there is no distinction between mutable and immutable arguments: `xsp` could potentially be updated by any function it is passed to.

The type inferred for a Pawns variable can be made less general by explicitly adding a type annotation, for example, the binding of `ysp` below. This allows it to be updated with a value of the less general type. A variable with a polymorphic type can potentially share with a variable with a less general type. The sharing analysis of Pawns uses type information to determine where this may occur and imposes an extra restriction. The variable with the more general type must not be modified as a side effect of a later statement (it must not be in a "!" annotation attached to a later statement). For example, in the code below the Pawns compiler accepts the direct update of `ysp` but the additional "`!xsp`" annotation results in an error when sharing analysis detects sharing between the two variables.

```
    *xsp = []              -- xsp has polymorphic type Ref [t]
    ysp = xsp::Ref [Int]   -- ysp has type Ref [Int]
    (assign !ysp [3]) !xsp -- update of ysp is OK but xsp is not!
    a1 = and *xsp          -- Problem: and applied to [Int]
```

Pawns supports polymorphic functions and variables in type signatures can be annotated with "!" but no sharing can be declared between parameters whose types are distinct type variables. If there is sharing between such arguments a separately named function with a more specific type signature and sharing declaration must be used (sharing is not polymorphic to the same extent as types). Unannotated type variables are considered abstract, and sharing is implicitly allowed in this case. To avoid duplicating function definitions, Pawns allows renaming of sets of functions. For example, we may have a polymorphic `map` function defined in terms of `foldr`. We can define versions for which more specific types and sharing can be declared as follows, with checking done as if the function definitions are duplicated and renamed:

```
renaming
    my_foldr_inst = foldr
    my_map_inst = map
```

11 Higher order programming

Like polymorphism, higher order code supports code reuse and abstraction, but combining it with destructive update requires great care. Pawns augments function types with information about mutability, sharing (including abstraction) and state variables. Each type `a->b` has additional information corresponding to "!" annotations and the pre- and post-condition (which has explicit sharing with `abstract` where appropriate). For a higher order function with type signature `(a->b)->c->d` there can be annotations and sharing information within the parentheses. Alternatively, a separate type can be defined, as below:

```
type Cords = [Cord]
type Cord_op = Cord -> Cord -> Cord
        sharing f c0 c1 = c
        pre nosharing
        post c = Branch c0 c1 -- result can share with both args

-- Haskell zipWith for cords with particular sharing
zip_with_cord :: Cord_op -> Cords -> Cords -> Cords
  sharing zip_with_cord f cs0 cs1 = cs
    pre nosharing
    post  -- elements of result share with elements of both args
        cs0 = Cons e0 _
        cs1 = Cons e1 _
        c = Cons e0 (Cons e1 _)
```

```
...
    zcs = zip_with_cord cord_app xcs ycs
```

Sharing analysis of the `zip_with_cord` definition treats the formal parameter `f` in the same way as a global function with the same sharing declared (when it is called, the precondition must be satisfied and the postcondition is added, etc). For code which calls `zip_with_cord`, the sharing declared in the precondition of its first argument (`cord_app` in the example above) must be a non-strict superset of that in the precondition of `f` and sharing in the pre+post-condition of this argument must be a non-strict subset of that in the pre+post-condition of `f`. Also, no argument in the type signature of `cord_app` can be annotated with "!". In general, a formal parameter must have a "!" in its type signature wherever the corresponding actual parameter has a "!". These rules are enforced during type checking rather than sharing analysis and also apply when functions are returned from functions or put into data structures.

Technically (and briefly), type checking generally uses a partial order over types: the notion of one (polymorphic) type being as general as another. Unification of types gives the greatest lower bound in this ordering. Pawns uses a partial order over annotated types. Annotated types are primarily ordered on their unannotated counterparts, and annotations are treated as follows. Given two annotated types t_1 and t_2 which are identical except for the annotations, t_1 is as general as t_2 if they are both non-arrow types and the arguments of t_1 are pairwise as general as those of t_2 or $t_1 = (a_1 \to b_1)$ with pre- and post-conditions c_1 and d_1 respectively, $t_2 = (a_2 \to b_2)$ with pre- and post-conditions c_2 and d_2 respectively, b_1 is as general as b_2, a_2 is as general as a_1, any mutable argument in t_1 is mutable in t_2, $c_1 \supseteq c_2$ and $c_2 \cup d_2 \supseteq c_1 \cup d_1$. In addition, the set of implicit `wo` state variables in t_1 must be the same as that in t_2, all implicit `rw` state variables in t_2 must be `rw` in t_1 and all implicit `ro` state variables in t_2 must be `ro` or `rw` in t_1.

Some additional complexities arise for sharing analysis with partial application since partial applications can contain shared (and mutable) data. It may seem diabolical that an assignment to a first order variable can mutate another variable which is a function, but the philosophy of Pawns is to permit such things as long as they are made obvious, as in the following:

```
const :: a -> b -> a
    sharing const a0 b0 = a1
    pre nosharing
    post a1 = a0
const a _ = a
...
    *xsp = [42]
    f = const (*xsp)       -- f is a function returning [42]
    np = lastp xsp         -- get ptr to [] in list *xsp
    *!np := [43] !xsp!f    -- mutate xsp and function f
    xs = f 0               -- f now returns [42,43]
```

The unannotated type of `const` can be written `a->(b->a)`. Without parentheses, the sharing information declared is attached to the second (inner) arrow: `b -> a sharing fn a0 b0 = a1 pre nosharing post a1 = a0`. This, with `a` replaced by `[Int]` is the annotated type of `f` in the code above. Note that the sharing information is defined using an equation with a dummy function name and two arguments even though the function has only one argument (of type `b`). The additional argument, `a0` represents an argument which is in a closure. In general, a function of type `a->b` may be a closure with any number of arguments already supplied. Sharing information for these arguments can be expressed by using additional arguments on the left side of the equation. Types for these arguments are inferred from outer arrows, or explicitly declared. Additional arguments in closures which are not declared in the type can be of any type and can have any sharing in the precondition. Sharing analysis uses special components of function variables to represent arguments in closures. For example, when `f` is bound in the code above, a component corresponding to the first closure argument shares with `xsp`. The postcondition of `lastp` causes sharing between (components of) `xsp` and `np` to be added, and therefore between `f` and `np`, so when `np` is assigned, we must add the annotation indicating both `xsp` and `f` can be updated.

The outer arrow in the type signature for `const` implicitly contains information about the sharing between the first argument and the closure. It can be made explicit as follows. Applying the keyword `closure` to one or more arguments indicates sharing of closure arguments and such postconditions are added by default to non-inner-most arrow types:

```
const :: a -> (b->a sharing fn a0 b0=a1 pre nosharing post a1=a0)
  sharing const a0 = c
  pre nosharing
  post c = closure a0
```

Another subtlety with sharing and closures is that functions which are "eta-equivalent" (and thus behave identically as pure functions) are not necessarily equivalent with respect to sharing. The functions definitions `maybenot` and `maybenot1` below are eta-equivalent but have different sharing declared and can behave differently when there is destructive update. In the following code, the closures `f` and `f1` are created. Because `maybenot` is defined with two arguments it is not evaluated at all at this stage and `f` is the closure `maybenot (Just True)`. A pointer to the argument of `Just` can be obtained and destructively updated, modifying `f`. In contrast, `maybenot1` is defined with a single argument and is evaluated, returning the closure `const True`. The closure argument is a constant, so it is not shared and cannot be destructively updated. The type signature of `maybenot` implicitly declares sharing with the closure argument, whereas `maybenot1` explicitly declares there is no such sharing.

```
maybenot :: Maybe Bool -> Bool -> Bool
maybenot m b =
    case m of
    (Just b1) -> const b1 b  -- return b1
    Nothing -> not b

maybenot1 :: Maybe Bool -> (Bool -> Bool)
    sharing maybenot1 m = r
    pre nosharing
    post nosharing       -- closure returned has no sharing
maybenot1 m =
    case m of
    (Just b1) -> const b1
    Nothing -> not

...
    m = Just True
    f = maybenot m       -- f is closure maybenot (Just True)
    f1 = maybenot1 m     -- f1 is closure const True
    case m of
    (Just *bp) ->
        (*!bp := False) !f -- smashes f but not f1
```

Explicitly declaring sharing for closures (and with abstract) is not normally needed due to the default processing, but we use the explicit representation in the sharing analysis and there are situations where the defaults are not quite what is desired.

12 Implementation status and other language features

We currently have a basic prototype version of Pawns implemented in Prolog, with several aspects of the design guided by ease of implementation. The syntax we support for Pawns is based on Prolog operator declarations rather than the Haskell-like syntax used in this paper (we have not written a parser or made a final decision on what syntax to support but the contents of the file before the first blank line is reserved for a Unix `#!` line, meta-data, syntax directives etc). Much of our effort has been devoted to sharing and mutability analysis as it is at the core of the novel aspects of Pawns. Programs are transformed into a relatively simple core language and arrow types are expanded so sharing, mutability and state variables are explicit for every arrow. We support a simple source language which is sufficient for writing and testing the analysis of small programs but lacks many things one might expect. For example, there is very limited support for built-in types and functions and the error messages leave a lot to be desired. As well as these more obvious deficiencies, many more features must be supported to make Pawns an appealing practical language. Here we briefly discuss initial performance results and some other language features (some of which are tentative).

The compiler performs a reasonably simple translation to C extended with macros and functions derived from algebraic data type definitions (this extension is implemented separately and is is similar to the ADT tool [11] but has more efficient data representation using tagged pointers). This allows a simple interface to C code which can use the same macros and functions. The bodies of Pawns function definitions can be quoted C code or "`extern`" (allowing linking with separately compiled C code); this nullifies any guarantee of purity etc suggested by type signatures but makes up for lack of features in Pawns itself. We use the Boehm-Demers-Weiser conservative garbage collector [12].

We have investigated performance of our version of the binary search tree insertion code presented in Section 6 using `gcc 4.6.3-1ubuntu5` running on a Dell laptop (`x86_64`). Although the generated C code appears very inefficient (a switch rather than an if-then-else, recursion instead of a while loop, etc), `gcc -O3` produces efficient iterative assembly code. The inner loop of `bst_insert_du` is around twice as fast as our hand-written iterative C coding, which we put down to the vagaries of modern optimizing compilers and hardware. For purely declarative version, `bst_insert_pure`, the resulting C code constructs a new tree node after the recursive call so `gcc` is unable to remove the (non-tail) recursion and it is slower by a factor of around twenty. The vast majority of this slowdown seems to be due to the call to `GC_malloc` in the inner loop — inserting a redundant `tmp = Node Empty x Empty` into the fast code causes a very similar slowdown. Thus more complex translation to C which makes recursion easier to avoid is unlikely to significantly improve the speed.

Pawns has a simple module system. A module can import/export types and functions from/to other modules. When exporting, the data constructors of a type can be hidden. Arrays of fixed size are a straightforward extension to algebraic data types. An array of type `t` can be seen as a data type with a distinct constructor for each array size, each with the appropriate number of arguments of type `t`. In Pawns, the arguments are (implicit) references to `t` (the ref view of the type) which can be obtained to update elements. We have implemented a small set of primitives: creating an array of a given size with some initialisation of elements, returning a pointer to an element given an array and index and returning the size of an array. A more extensive set of array primitives is planned, along with some syntactic sugar. Given that destructive update is supported, it seems convenient to make some of the typical control constructs of imperative languages available, such as while and for loops. Such loop constructs can be translated into recursive functions, but loop constructs are typically more concise and many programmers prefer them.

For low level manipulation of data structures containing pointers, a form of pointer equality is often needed[8]. Although functions which use pointer equality may not perform destructive update, they cannot be considered "pure" because their result depends on the low level representation of values rather than the

[8] Like the ghc primitive `reallyUnsafePtrEquality# :: a -> a -> Int#`

values themselves. The simplest way to incorporate pointer equality into Pawns is for the type signature of the pointer equality primitive to declare that the arguments may be updated, using "!" annotations. This is overkill and we are considering a more refined method of declaring impurity which distinguishes the result of a function depending on the low level "store" view of data from whether the data may be updated. With our current design, "?" annotations imply the low level view can be used, and may be needed for verification of the whole program, but it does not imply the function can be impure.

It may also be convenient to support assertions concerning (non)sharing, which the sharing analysis assumes are correct, leading to greater precision. Such assertions could optionally be checked at runtime. For low level code, imprecision of sharing analysis leads to redundant "!" annotations at worst, which is not a great cost for the comfort of compiler-proven properties. However, imprecision can also prevent some mixing of high and low level code, forcing the programmer to make more code low level. If experience shows that this is particularly burdensome, it may warrant the introduction of assertions.

13 Related programming languages

Many declarative languages are designed and/or implemented so destructive update can be performed on data structures which are not shared. This is particularly important for arrays, with $O(1)$ time for access and update of elements, which are used in many important algorithms. Various language features have been used to ensure data structures are single threaded though the computation and hence not shared, for example, monads in Haskell [8], unique types in Clean[9] [13] and unique modes in Mercury[10] [14]. Sharing analysis of the kind used in Pawns is performed in the Mars language (including analysis of higher order code) to check for single threading and is also used in Mercury for compile time garbage collection and structure re-use.

For single threaded data structures, Pawns provides the syntactic convenience of state variables. Mercury supports some syntactic sugar for passing extra values in and out of procedures but still requires calls and definitions to be modified. Haskell's monadic "do" notation requires even more code modification, and is particularly cumbersome when access to and modification of multiple data structures (or IO) is required. Pawns state variables give the convenience of global variables while only requiring a "!" prefix on calls and additional information in function prototypes declaring what variables are introduced, used and modified. However, a more fundamental distinguishing feature of Pawns is that it allows destructive update of *shared* structures — single threading is not required. For data structures which are single threaded but not implemented using state variables, Pawns code typically has "!" annotations just on arguments of function calls and the left side of assignments; additional annotations on statements indicate that updated data structures are not single threaded.

[9] `http://wiki.clean.cs.ru.nl/Clean`
[10] `http://mercurylang.org/`

Despite the focus on update of shared structures, Pawns has some other similarities with Mars and Mercury. The way Pawns uses sharing with `abstract` to prevent destructive update of some data structures is similar to the way Mars protects "static" values from being updated. However, in Mars, such values are effectively shared with a special variable when they are created and can never be updated in any context. In Pawns, possible sharing with the special variable is an artifact of the sharing analysis which varies with the context. A data structure can be created in a context where is doesn't share with `abstract`, and can be updated, but in other contexts update may be blocked. The purity system of Mercury allows code which does destructive update to be encapsulated in pure code, though this is done using a "promise" from the programmer rather than checks done by the compiler. Mercury has "impure" code, which can perform side effects such as destructive update as well as "semipure" code has no effects but the result may be dependent on some context (such as a global variable) rather than just being dependent on the arguments. A semipure Mercury function is not a function in the mathematical sense. For higher order arguments the purity is fixed, resulting in the need to sometimes duplicate code for higher order functions. Pawns has impure functions and divides pure functions into two categories — those where low level details such as sharing should be considered and those where it should not. For pure functions where sharing is declared, the high level view does not give the whole story. Other parts of the program may rely on the representation of the values computed rather than just the values themselves.

There are other (primarily) declarative languages with the ability to update shared structures. 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 operations on the store can be serialised and functions can be considered "pure". 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. There is no implicit (low level) view of all types which include refs as is supported in Pawns, so there can be multiple variants of data types and less efficient data representation as discussed in Section 4. Furthermore, when a ref data type is passed to a function call, it is not clear from the code what variables may be updated — it is similar to the situation in imperative languages. For example, consider the Haskell `DUTree` type and `dubst_size` function given earlier. The data type must contain `STRef`'s so that the tree can be destructively updated, complicating the code and introducing an extra level of indirection in the representation. Although `dubst_size` does not perform any assignments, a different definition with the same interface (type signature) could update its argument and/or any other variable in the `ST s` monad. Mutability is tied to the ref type and there is no additional mechanism to identify what variables will be updated (if any). This contrasts with Pawns which allows mutability of any type (with a non-zero arity data constructor) but has a separate mechanism for annotating individual variables when they may be updated. Consistent with a mutable data structure and imperative style code, the "do" notation in the Haskell code explicitly sequences the dereference operations and recursive calls, resulting in much more verbose code than the Pawns `bst_size` function.

If we take the "ref" view of Pawns code, it is semantically very similar to ML. Only the argument of the `Ref` data constructor can be updated (its just that `Ref` data constructors are in every non-atomic type). However, there is a significant difference in implementation. In Pawns, a `Ref` in the argument of a data constructor is optimised away, so unlike `ref` in ML and `STRef` in Haskell, no indirection is used in the representation (in OCAML mutable fields of records can be used to avoid indirection, but only if the field does not have a polymorphic type). This can be done very easily due to two features of (the ref view of) Pawns. First, every data constructor argument is a `Ref` (except arguments of `Ref` itself). Second, each such `Ref` is unique: whenever a data constructor is applied, `Ref` is implicitly applied to each argument separately so it is not possible to have an identical `Ref` in two distinct arguments. In normal pattern matching the pattern variables are bound to the argument of each (implicit) `Ref`. In dereference patterns, variables are bound to `Ref`s, and this can be done simply by using the address of the word where the argument is stored, rather than the contents of the address (we use the l-value rather than the r-value). Taking the high level view instead of the ref view, dereference patterns create a ref. It would be possible to expose the `Ref` data constructor in the Pawns language, but we feel a mixture of implicit and explicit `Ref`s would be very confusing. Hence we have avoided using the `Ref` data constructor in Pawns and support dereferencing on the left of `=` as an (equally expressive) alternative.

Like Pawns, the Disciple language [15] supports update of arguments of all data constructors rather than a distinguished "ref" data constructor. Disciple uses "region" information [16] in the type system, which is similar to the transitive closure of the sharing information used in Pawns. For example, for a set of variables which are lists, the region information partitions the set according to which region their cons cells occupy. Sharing information can express the fact that `x` may share with `y` and `z` but `y` and `z` do not share with each other, so in this sense region information is less expressive than sharing. For N variables, the number of different ways of partitioning into regions is given by the Bell (or exponential) numbers[11]: 1, 2, 5, 15, 52, 203, ... The number of different ways of sharing is $2^{N(N-1)/2}$: 1, 2, 8, 64, 1024, 32768, ...

The non-transitivity of sharing information seems important for pre- and post-conditions. 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 correctness of the cord code. This precondition cannot be expressed with region information — the regions for all the variables must be the same due to transitivity. Sharing with `abstract` in Pawns also helps with reasoning about program correctness and has no analogue in Disciple.

[11] `http://www.research.att.com/~njas/sequences/A000110`

However, there is also information which can be expressed in Disciple but not in Pawns. The regions of Disciple have several possible attributes such as being able to be destructively updated. Unfortunately, such information about destructive update is currently not encapsulated inside function definitions. For example, in the equivalent of our code for `list_bst`, the region containing the tree nodes must be mutable for `bst_insert_du` to perform the update, but that information leaks out in the type signature of `list_bst`. This leakage of information, plus the fact that the compiler can infer information about destructive update (rather than it being declared) can mean it is less clear what variables can be changed by an assignment or call to an impure function. Polymorphism and higher order are supported with destructive update, at the cost of a rather complex type system.

There are also primarily imperative languages which support some features of declarative languages. For example, the Rust language[12] is designed to make pointers safe and it has some support for algebraic data types. The way ADTs are supported in Rust exposes the fact that pointers are used, and generally results in a less efficient representation (in common with the ADT tool). Pawns, in common with other declarative languages, avoids exposing the fact that (possibly tagged) pointers are used. This allows significantly more efficient data representations for data types which have more than one data constructor with arguments. A slight inconvenience is that we can only perform destructive update on arguments of data constructors. This significantly influences how destructive update can be incorporated into the language.

The way Pawns and other declarative languages support algebraic data types does have some efficiency penalties however. It is hard to avoid using pointers to heap-allocated data for non-atomic values. In most imperative languages, such values can be allocated on the stack, reducing memory management overheads. For large objects such as arrays the overhead is small in percentage terms, but for a type such as a pair of integers, allocation on the stack may be significantly better. A straightforward implementation of Pawns does not allow pointers to the stack. This avoids possible dangling pointers but means that all destructive update is done on the heap rather than the stack. There are opportunities for optimising some cases (for example, we compile state variables to static pointers), but it is likely a performance penalty will remain compared to the fastest imperative languages.

14 Conclusion

There are important algorithms which rely on 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. By viewing algebraic data types at different levels of abstraction, Pawns allows pointers to arguments of data constructors which can be used for destructive update of shared data structures. Pawns includes

[12] `http://www.rust-lang.org/`

several features which allow these effects to be encapsulated, so the 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 post-conditions which describe the sharing of arguments and the result when the function is called and when it returns. To avoid having to consider sharing of data structures for all the code, some function arguments and results can be declared abstract. 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 result from function calls.

Although Pawns is still in the early stages of development, and may never 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 destructive update of shared data structures.

Acknowledgements

The design of Pawns has benefitted from discussions with many people. Bernie Pope and Peter Schachte particularly deserve a mention.

References

1. Warren, D.H.D.: An abstract Prolog instruction set. Tecnical Note 309, SRI International, Menlo Park, California (October 1983)
2. Taylor, A.: Parma - bridging the performance gap between imperative and logic programming. J. Log. Program. **29**(1-3) (1996) 5–16
3. Milner, R., Tofte, M., Macqueen, D.: The Definition of Standard ML. MIT Press, Cambridge, MA, USA (1997)
4. Chen, C.P., Hudak, P.: Rolling your own mutable adt—a connection between linear types and monads. In: Proceedings of 24th ACM Symposium on Principles of Programming Languages, New York, ACM Press (January 1997) 54–66
5. Mulkers, A.: Live data structures in logic programs, derivation by means of abstract interpretation. Springer-Verlag (1993)

6. Mazur, N., Ross, P., Janssens, G., Bruynooghe, M.: Practical aspects for a working compile time garbage collection system for Mercury. In Codognet, P., ed.: Proceedings of ICLP 2001, Lecture Notes in Computer Science. Volume 2237., Springer (2001) 105–119
7. Giuca, M.: Mars: An imperative/declarative higher-order programming language with automatic destructive update. PhD thesis, University of Melbourne (To appear, 2014)
8. Peyton Jones, S., Wadler, P.: Imperative functional programming. In: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (1993) 71–84
9. Reynolds, J.C.: An overview of separation logic. In Meyer, B., Woodcock, J., eds.: VSTTE. Volume 4171 of Lecture Notes in Computer Science., Springer (2005) 460–469
10. Wright, A.: Simple imperative polymorphism. In: LISP and Symbolic Computation. (1995) 343–356
11. Hartel, P.H., Muller, H.L.: Simple algebraic data types for C. Softw., Pract. Exper. **42**(2) (2012) 191–210
12. Boehm, H.J., Weiser, M.: Garbage collection in an uncooperative environment. Softw. Pract. Exper. **18**(9) (September 1988) 807–820
13. Plasmeijer, R., van Eekelen, M.: Concurrent Clean language report version 2.1. University of Nijmegen, November (2002)
14. Somogyi, Z., Henderson, F., Conway, T.C.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. J. Log. Program. **29**(1-3) (1996) 17–64
15. Lippmeier, B.: Type Inference and Optimisation for an Impure World. PhD thesis, Australian National University (June 2009)
16. Talpin, J.P., Jouvelot, P.: Polymorphic type, region and effect inference. Journal of Functional Programming **2** (7 1992) 245–271