Sharing analysis in the Pawns compiler

Lee Naish

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

Abstract. Pawns is a programming language under development which supports algebraic data types, polymorphism, higher order functions and "pure" declarative programming. It also supports impure imperative features including destructive update of shared data structures via pointers, allowing significantly increased efficiency for some operations. A novelty of Pawns is that all impure "effects" must be made obvious in the source code and they can be safely encapsulated in pure functions in a way that is checked by the compiler. Execution of a pure function can perform destructive updates on data structures which are local to or eventually returned from the function without risking modification of the data structures passed to the function. This paper describes the sharing analysis which allows impurity to be encapsulated. Aspects of the analysis are similar to other published work, but in addition it handles explicit pointers and destructive update, higher order functions including closures and pre- and postconditions concerning sharing for functions.
Keywords: functional programming language, destructive update, mutability, effects, algebraic data type, sharing analysis, aliasing analysis

1 Introduction

This paper describes the sharing analysis done by the compiler for Pawns [1], a programming language which is currently under development. Pawns supports both declarative and imperative styles of programming. It supports algebraic data types, polymorphism, higher order programming and "pure" declarative functions, allowing very high level reasoning about code. It also allows imperative code, where programmers can consider the representation of data types, obtain pointers to the arguments of data constructors and destructively update them. Such code requires the programmer to reason at a much lower level and consider aliasing of pointers and sharing of data structures. Low level "impure" code can be encapsulated within a pure interface and the compiler checks the purity. This requires analysis of pointer aliasing and data structure sharing, to distinguish data structures which are only visible to the low level code (and are therefore safe to update) from data structures which are passed in from the high level code (for which update would violate purity). The main aim of Pawns is to get the benefits of purity for most code but still have the ability to write some key components using an imperative style, which can significantly improve efficiency (for example, a more than twenty-fold increase in the speed of inserting an element into a binary search tree).

There are other functional programming languages, such as ML [2], Haskell [3] and Disciple [4], which allow destructive update of shared data structures but do not allow this impurity to be encapsulated. In these languages the ability to update the data structure is connected to its type[1]. For a data structure to be built using destructive update its type must allow destructive update and any code which uses the data structure can potentially update it as well. This prevents simple declarative analysis of the code and can lead to a proliferation of different versions of a data structure, with different parts being mutable. There is often an efficiency penality as well, with destructive update requiring an extra level of indirection in the data structure. Pawns avoids this inefficiency and separates mutablity from type information, allowing a data structure to be mutable in some contexts and considered "pure" in others. The main cost from the programmer perspective is the need to include extra annotations and information in the source code. This can also be considered a benefit, as they provide useful documentation and error checking. The main implementation cost is additional analysis done by the compiler, which is the focus of this paper.

The rest of this paper assumes some familiarity with Haskell and is structured as follows. Section 2 gives a brief overview of the relevant features of Pawns and Section 3 describes a simple "core" language which source programs are translated into. Section 4 describes the abstract domain used for sharing analysis algorithm, Section 5 defines the algorithm itself and Section 6 gives an extended example. Section 7 briefly discusses precision and efficiency issues. Section 8 discusses related work and Section 9 concludes.

2 An overview of Pawns

A more detailed introduction to Pawns is given in [1]. Pawns has many similarities with other functional languages. It supports algebraic data types with parametric polymorphism, higher order programming and curried function definitions. It uses strict evaluation. In addition, it supports destructive update via "references" (pointers) and has a variety of extra annotations to make impure effects more clear from the source code and allow them to be encapsulated in pure code. Pawns also supports a form of global variables (called state variables) which support encapsulated effects, but we do not discuss them further here as they are handled in essentially the same way as other variables in sharing analysis. Pure code can be thought of in a declarative way, were values can be viewed abstractly, without considering how they are represented. Code which uses destructive update must be viewed at a lower level, considering the representation of values, including sharing. We discuss this lower level view first, then briefly

[1] Disciple uses "region" information to augment types, with similar consequences.

present how impurity can be encapsulated to support the high level view. We use Haskell-like syntax for familiarity.

2.1 The low level view

Values in Pawns are represented as follows. Constants (data constructors with no arguments) are represented using a value in a single word. A data constructor with $N > 0$ arguments is represented using a word that contains a tagged pointer to a block of N words in main memory containing the arguments. For simple data types such as lists the tag may be empty. In more complex cases some bits of the pointer may be used and/or a tag may be stored in a word in main memory along with the arguments. Note that constants and tagged pointers are not always stored in main memory and Pawns variables may correspond to registers that contain the value. Only the arguments of data constructors are guaranteed to be in main memory. An array of size N is represented in the same way as a data constructor with N arguments, with the size given by the tag. Functions are represented as either a constant (for functions which are known statically) or a closure which is a data constructor with a known function and a number of other arguments.

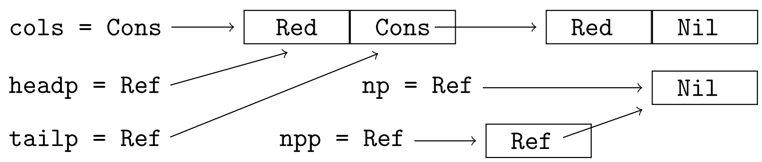
Pawns has a `Ref t` type constructor, representing a reference/pointer to a value of type `t` (which must be stored in memory). Conceptually we can think of a corresponding `Ref` data constructor with a single argument, but this is never explicit in Pawns code. Instead, there is an explicit dereference operation: `*vp` denotes the value `vp` points to. There are two ways references can be created: let bindings and pattern bindings. A let binding `*vp = val` allocates a word in main memory, initializes it to `val` and makes `vp` a reference to it (Pawns omits Haskell's `let` and `in` keywords; the scope is the following sequence of statements/expressions). In a pattern binding, if `*vp` is the argument of a data constructor pattern, `vp` is bound to a reference to the corresponding argument of the data constructor if pattern matching succeeds (there is also a primitive which returns a reference to the i^{th} element of an array). Note it is not possible to obtain a reference to a Pawns variable: variables do not denote memory locations. However, a variable `vp` of type `Ref t` denotes a reference to a memory location containing a value of type `t` and the memory location can be destructively updated by `*vp := val`.

Consider the following code. Two data types are defined. The code creates a reference to `Nil` (`Nil` is stored in a newly allocated memory word) and a reference to that reference (a pointer to the word containing `Nil` is put in another allocated word). It also creates a list containing constants `Blue` and `Red` (requiring the allocation of two cons cells in memory; the `Nil` is copied). It deconstructs the list to obtain pointers to the head and tail of the list (the two words in the first cons cell) then destructively updates the head of the list to be `Red`.

```
data Colour = Red | Green | Blue
data Colours = Nil | Cons Colour Colours -- like List Colour
```

```
    ...
    *np = Nil                          -- np = ref to (copy of) Nil
    *npp = np                          -- npp = ref to (copy of) np
    cols = Cons Blue (Cons Red *np) -- cols = [Blue, Red]
    case cols of
    (Cons *headp *tailp) ->            -- get ref to head and tail
        *headp := Red                  -- update head with Red
```

The memory layout after the assignment can be pictured as follows, where boxes represent main memory words and `Ref` and `Cons` followed by an arrow represent pointers (no tag is used in either case):



The destructive update above changes the values of both `headp` and `cols` (the representations are shared). One of the novel features of Pawns is that the source code must be annotated with "!" to make it obvious when each "live" variable is updated. If both `headp` and `cols` are used later, the assignment statement above must be written as follows, with `headp` prefixed with "!" and an additional annotation attached to the whole statement indicating `cols` may be updated:

```
        *!headp := Red  !cols        -- update *headp (and cols)
```

We say that the statement *directly* updates `headp` and *indirectly* updates `cols`, due to sharing of representations. Similarly, if `headp` was passed to a function which may update it, additional annotations are required. For example, `(assign !headp Red) !cols` makes the direct update of `headp` and indirect update of `cols` clear. Sharing analysis is used to ensure that source code contains all the necessary annotations. One aim of Pawns is that any effects of code should be made clear by the code. Pawns is an acronym for Pointer Assignment With No Surprises.

Pawns functions have extra annotations in type signatures to document which arguments may be updated. For additional documentation, and help in sharing analysis, there are annotations to declare what sharing may exist between arguments when the function is called (a precondition) and what extra sharing may be added by executing the function (called a postcondition, though it is the union of the pre- and post-condition which must be satisfied after a function is executed). For example, we may have:

```
assign :: Ref t -> t -> ()
    sharing assign !p v = _    -- p may be updated
    pre nosharing              -- p&v don't share when called
    post *p = v                -- assign may make *p alias with v
```

As well as checking for annotations on assignments and function calls, sharing analysis is used to check that all arguments which may be updated are annotated in type signatures, and pre- and post-conditions are always satisfied. For example, assuming the previous code which binds `cols`, the call `assign !tailp !cols` annotates all modified variables but violates the precondition of `assign` because there is sharing between `tailp` and `cols` at the time of the call. Violating this precondition allows cyclic structures to be created, which is important for understanding the code. In general, there is an inter-dependence between "!" annotations in the code and pre- and post-conditions. More possible sharing at a call means more "!" annotations are needed, more sharing in (recursive) calls and more sharing when the function returns.

Curried functions and higher order code are supported by attaching sharing and destructive update information to each arrow in a type, though often the information is inferred rather than being given explicitly in the source code. For example, implicit in the declaration for `assign` above is that `assign` called with a single argument of type `Ref t` creates a closure of type `t -> ()` containing that argument (and thus sharing the object of type `t`). The explicit sharing information describes applications of this closure to another argument. There is a single argument in this application, referred to with the formal parameter `v`. The other formal parameter, `p`, refers to the argument of the closure. In general, a type with N arrows in the "spine" has $K+N$ formal parameters in the description of sharing, with the first K parameters being closure arguments.

The following code defines binary search trees of integers and defines a function which takes a pointer to a tree and inserts an integer into the tree. It uses destructive update, as would normally be done in an imperative language. The declarative alternative must reconstruct all nodes in the path from the root down to the new node. Experiments using our prototype implementation of Pawns indicate that for long paths this destructive update version is as fast as hand-written C code whereas the "pure" version is more than twenty times slower, primarily due to the overhead of memory allocation.

```
data Tree = Empty | Node Tree Int Tree

bst_insert_du :: Int -> Ref Tree -> ()
    sharing bst_insert_du x !tp = _   -- tree gets updated
    pre nosharing                     -- integers are atomic so
    post nosharing                    -- it doesn't share
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 -- update lp (and tp)
        else
            (bst_insert_du x !rp) !tp -- update rp (and tp)
```

2.2 The high level view

Whenever destructive update is used in Pawns, programmers must be aware of potential sharing of data representations and take a low level view. In other cases it is desirable to have a high level view of values, ignoring how they are represented and any sharing which may be present. Pawns has a mechanism to indicate that such a high level view is taken. Pre- and post-conditions can specify sharing with a special pseudo-variable named `abstract`[2]. No variables which share with `abstract` can be destructively updated. Pawns type signatures which have no annotations concerning destructive update or sharing implicitly indicate no arguments are destructively updated and the arguments and result share with `abstract`. Thus a subset of Pawns code can look like and be considered as pure functional code.

The following code defines a function which takes a list of integers and returns a binary search tree containing the same integers. Though it uses destructive update internally, this impurity is encapsulated and it can therefore be viewed as a pure function. The list which is passed in as an argument is never updated. An initially empty tree is created locally. It is destructively updated by inserting each integer of the list into it (using `list_bst_du`, which calls `bst_insert_du`), then the tree is returned. Within the execution of `list_bst` it is important to understand the low level details of how the tree is represented, but this information is not needed outside the call. The sharing analysis of the Pawns compiler allows a distinction between "abstract" variables, which cannot be updated, and "concrete" variables which can be updated. Sharing of concrete variables must be considered and explicitly documented by the programmer.

```
data Ints = Nil | Cons Int Ints

list_bst :: Ints -> Tree  -- pure function from Ints to Tree
 -- implicit sharing information:
 -- sharing list_bst xs = t
 -- pre xs = abstract
 -- post t = abstract
list_bst xs =
    *tp = Empty           -- create pointer to empty tree
    list_bst_du xs !tp    -- insert integers into tree, updating it
    *tp                   -- return tree
```

[2] There is conceptually a different `abstract` variable for each distinct type.

```
list_bst_du :: Ints -> Ref Tree -> ()
    sharing list_bst_du xs !tp = _  -- tree gets updated
    pre xs = abstract
    post nosharing
list_bst xs =
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 -> ()
```

3 Core Pawns

An early pass of the Pawns compiler converts all function definitions into a core language by flattening nested expressions, introducing extra variables et cetera. A variable representing the return value of the function is introduced and expressions are converted to bindings for variables. A representation of the core language version of code is annotated with type, liveness and other information prior to sharing analysis. We just describe the core language here. The right side of each function definition is a statement (described using the definition of type `Stat` below), which may contain variables, including function names (`Var`), data constructors (`DCons`) and pairs containing a pattern (`Pat`) and statement for case statements. All variables are distinct except for those in recursive instances of `Stat` and variables are renamed to avoid any ambiguity due to scope.

```
data Stat =                   -- Statement, eg
    Seq Stat Stat |           -- stat1 ; stat2
    EqVar Var Var |           -- v = v1
    EqDeref Var Var |         -- v = *v1
    DerefEq Var Var |         -- *v = v1
    DC Var DCons [Var] |      -- v = Cons v1 v2
    Case Var [(Pat, Stat)] |  -- case v of pat1 -> stat1 ...
    Error |                   -- (for uncovered cases)
    App Var Var [Var] |       -- v = f v1 v2
    Assign Var Var |          -- *!v := v1
    Instype Var Var           -- v = v1::instance_of_v1_type

data Pat =                    -- patterns for case, eg
    Pat DCons [Var]           -- (Cons *v1 *v2)
```

Patterns in the core language only bind references to arguments — the arguments themselves must be obtained by explicit dereference operations. Pawns supports “default” patterns but for simplicity of presentation here we assume all patterns are covered in core Pawns and we include an error primitive. Similarly, we just give the general case for application of a variable to $N > 0$ arguments;

our implementation distinguishes some special cases. Memory is allocated for `DerefEq`, `DC` (for non-constants) and `App` (for unsaturated applications which result in a closure).

Sharing and type analysis cannot be entirely separated. Destructive update in the presence of polymorphic types can potentially violate type safety or "preservation". For a variable whose type contains a type variable, we must avoid destructive update with a value with a less general type. For example, in `*x = []` the type of `x` is `Ref [t]`. If `*x` is assigned `[42]`, of type `[Int]`, passing it to a function which expects a `[Bool]` violates type safety. Pawns allows expressions to have their inferred types further instantiated using "::", and the type checking pass of the compiler also inserts some type instantiation. The type checking pass ensures that direct update does not involve type instantiation but to improve flexibility, indirect update is checked during the sharing analysis.

4 The abstract domain

The representation of the value of a variable includes some set of main memory words (arguments of data constructors). Two variables share if the intersection of their sets of main memory words is not empty. The abstract domain for sharing analysis must maintain a conservative approximation to all sharing, so we can tell if two variables possibly share (or definitely do not share). The abstract domain we use is a set of pairs (representing possibly intersecting sets of main memory locations) of variable *components*. The different components of a variable partition the set of main memory words for the variable.

The components of a variable depend on its type. For non-recursive types other than arrays, each possible data constructor argument is represented separately. For example, the type `Maybe (Maybe (Either Int Int))` can have an argument of an outer `Just` data constructor, an inner `Just` and `Left` and `Right`. A component can be represented using a list of `x.y` pairs containing a data constructor and an argument number, giving the path from the outermost data constructor to the given argument. For example, the components of the type above can be written as: `[Just.1]`, `[Just.1,Just.1]`, `[Just.1,Just.1,Left.1]` and `[Just.1,Just.1,Right.1]`. If variable `v` has value `Just Nothing`, the expression `v.[Just.1]` represents the single main memory word containing the occurrence of `Nothing`.

For `Ref t` types we proceed as if there was a `Ref` data constructor, so `vp.[Ref.1]` represents the word `vp` points to. For function types, values may be closures. A closure which has had K arguments supplied is represented as a data constructor $\texttt{Cl}_K$ with these K arguments; these behave in the same way as other data constructor arguments with respect to sharing. Closures also contain a code pointer and an integer which are not relevant to sharing so we ignore them here. We also ignore the subscript on the data constructor for sharing analysis because type and sharing analysis only give a lower bound on the number of closure arguments. Our analysis orders closure arguments so that the most recently supplied argument is first (the reverse of the more natural ordering).

For arrays, [Array_.1] is used to represent all words in the array. The expression, x.[Array_.1,Just.1] represents the arguments of all Just elements in an array x of Maybe values. For recursive types, paths are "folded" [5] so there are a finite number of components. If a type T has sub-component(s) of type T we use the empty path to denote the sub-component(s). In general, we construct a path from the top level and if we come across a sub-component of type T which is in the list of ancestor types (the top level type followed by the types of elements of the path constructed so far) we just use the path to the ancestor to represent the sub-component. Consider the following mutually recursive types:

```
data RTrees = Nil | Cons RTree RTrees
data RTree = RNode Int RTrees
```

For type RTrees we have the components [] (this folded path represents both [Cons.2] and [Cons.1,RNode.2], since they are of type RTrees), [Cons.1] and [Cons.1,RNode.1]. The expression t.[Cons.1,RNode.1] represents the set of memory words which are the first argument of RNode in variable t of type RTrees. For type RTree we have the components [] (for [RNode.2,Cons.1], of type RTree), [RNode.1] and [RNode.2] (which is also the folded version of [RNode.2,Cons.2], of type RTrees). In our sharing analysis algorithm we use a function fc (fold component) which takes a $v.c$ pair, and returns $v.c'$ where c' is the correctly folded component for the type of variable v. For example, fc (ts.[Cons.2]) = ts.[], assuming ts has type RTrees.

As well as containing pairs of components for distinct variables which may alias, the abstract domain contains "self-sharing" pairs for each possible component of a variable which may exist. Consider the following two bindings:

```
t = RNode 2 Nil
ts = Cons t Nil
```

With our domain, the most precise description of sharing after these two bindings is as follows. We represent a sharing pair as a set of two variable components. The first five are self-sharing pairs and the other two describe the sharing between t and ts.

```
{{t.[RNode.1], t.[RNode.1]},
 {t.[RNode.2], t.[RNode.2]},
 {ts.[], ts.[]},
 {ts.[Cons.1], ts.[Cons.1]},
 {ts.[Cons.1,RNode.1], ts.[Cons.1,RNode.1]},
 {t.[RNode.1], ts.[Cons.1,RNode.1]},
 {t.[RNode.2], ts.[]}}
```

Note there is no self-sharing pair for t.[] since there is no strict sub-part of t which is an RTree. Similarly, there is no sharing between ts.[Cons.1] and any part of t. Although the value t is used as the first argument of Cons in ts, this is not a main memory word which is used to represent the value of t (indeed, the value of t has no Cons cells). The tagged pointer value stored in variable t (which may be in a register) is copied into the cons cell.

5 The sharing analysis algorithm

We now describe the sharing analysis algorithm. Overall, the compiler attempts to find a proof that for a computation with a depth D of (possibly recursive) function calls, the following condition C holds, assuming C holds for all computations of depth less than D. This allows a proof by induction that C holds for all finite computations.

C: For all functions f, if the precondition of f is satisfied whenever f is called, then

1. for all function calls and assignment statements in f, any live variable that may be updated at that point in an execution of f is annotated with "!",
2. there is no update of live "abstract" variables when executing f,
3. the union of the pre- and post-conditions of f is satisfied when f returns,
4. all parameters of f which may be updated when executing f are declared mutable in the type signature of f,
5. for all function calls and assignment statements in f, any live variable that may be directly updated at that point is updated with a value of the same type or a more general type, and
6. for all function calls and assignment statements in f, any live variable that may be indirectly updated at that point does not share with any variable which has a less general type.

The algorithm is applied to each function definition in core Pawns to compute an approximation to the sharing before and after each statement (we call it the alias set). This can be used to check points 1–3 and 6 above and that preconditions of called functions are satisfied, so the induction hypothesis can be used. Point 4 is established using point 1 and a simple syntactic check that any parameter of f which is annotated "!" in the definition is declared mutable in the type signature (parameters are considered live throughout the definition). Point 5 relies on 4 and the type checking pass. The core of the algorithm is to compute the alias set after a statement, given the alias set before the statement. This is applied recursively for compound statements.

The alias set used at the start of the definition is the precondition of the function. This implicitly includes self-sharing pairs for all variable components of the arguments of the function and the pseudo-variables $\texttt{abstract}_T$ for each type T used. Similarly, the postcondition implicitly includes self-sharing pairs for all components of the result (and the $\texttt{abstract}_T$ variable if the result is abstract)[3]. As analysis proceeds, extra variables from the function body are added to the alias set and variables which are no longer live can be removed to improve efficiency. The alias set computed for the end of the definition, with sharing for local variables removed, must be a subset of the union of the pre- and post-condition of the function. We assume type information is given for all variables (a type checking/inference pass is completed before sharing analysis) and sharing

[3] Self-sharing for arguments and results is usually desired. For the rare cases it is not, we may provide a mechanism to override this default in the future.

information is given for all type instances of all (possibly polymorphic) defined functions. All type variables in type assignments in the function definition are replaced by `Ref ()`. This type has a single component which can be shared to represent possible sharing of arbitrary components of an arbitrary type. Finally, we assume there is no type which is an infinite chain of refs, for example, `type Refs = Ref Refs` (for which type folding results in an empty component rather than a `[Ref.1]` component; this is not a practical limitation).

Suppose a_0 is the alias set just before statement s. The following algorithm computes $\texttt{alias}(s, a_0)$, the alias set just after statement s. The algorithm structure follows the recursive definition of statements and we describe it using psuedo-Haskell, interspersed with brief discussion. At some points we use high level declarative set comprehensions to describe what is computed and naive implementation may not lead to the best performance.

```
alias (Seq stat1 stat2) a0 =              -- stat1; stat2
   alias stat2 (alias stat1 a0)
alias (EqVar v1 v2) a0 =                  -- v1 = v2
   let
      self1 = {{v1.c, v1.c} | {v2.c, v2.c} ∈ a0}
      share1 = {{v1.c1, v.c2} | {v2.c1, v.c2} ∈ a0}
   in
      a0 ∪ self1 ∪ share1
alias (DerefEq v1 v2) a0 =                -- *v1 = v2
   let
      self1 = {{v1.[Ref.1], v1.[Ref.1]}} ∪
               {{fc(v1.(Ref.1 :c)), fc(v1.(Ref.1 :c))} | {v2.c, v2.c} ∈ a0}
      share1 = {{fc(v1.(Ref.1 :c1)), v.c2} | {v2.c1, v.c2} ∈ a0}
   in
      a0 ∪ self1 ∪ share1
```

Sequencing is handled by function composition. To bind a fresh variable `v1` to a variable `v2` the self-sharing of `v2` is duplicated for `v1` and the sharing for each component of `v2` is duplicated for `v1`. Binding `*v1` to `v2` is done in a similar way, but the components of `v1` must have `Ref.1` prepended to them and the result folded, and the `[Ref.1]` component of `v1` self-shares.

```
alias (Assign v1 v2) a0 =                 -- *v1 := v2
   let
      self1 = {{v1.[Ref.1], v1.[Ref.1]}} ∪
               {{fc(v1.(Ref.1 :c)), fc(v1.(Ref.1 :c))} | {v2.c, v2.c} ∈ a0}
      share1 = {{fc(v1.(Ref.1 :c1)), v.c2} | {v2.c1, v.c2} ∈ a0}
      -- al = possible aliases for v1.[Ref.1]
      al = {va.ca | {v1.[Ref.1], va.ca} ∈ a0}
      -- (live variables in al+v1 must be annotated with !
      -- and must not share with abstract)
```

```
      selfal = {{fc(v_a.(c_a++c)), fc(v_a.(c_a++c))} |
                       v_a.c_a ∈ al ∧ {v2.c, v2.c} ∈ a0}
      shareal = {{fc(v_a.(c_a++c_1)), v.c_2} |
                       v_a.c_a ∈ al ∧ {v2.c_1, v.c_2} ∈ a0} ∪
                    {{fc(v_a.(c_a++c)), fc(v1.(Ref.1 :c))} |
                       v_a.c_a ∈ al ∧ {v2.c, v2.c} ∈ a0}
      -- old1 = old aliases for v1, which can be removed
      -- if the assignment doesn't create a cyclic structure
      old1 = {{v1.c_1, v.c_2} | {v1.c_1, v.c_2} ∈ a0}
   in if ∃c {v1.[Ref.1], v2.c} ∈ a0 then
         a0 ∪ self1 ∪ share1 ∪ selfal ∪ shareal
      else
         (a0 \ old1) ∪ self1 ∪ share1 ∪ selfal ∪ shareal
```

Assignment to an existing variable `*v1` adds the same sharing as for binding a fresh variable, but there are two extra complications. First, `*v1` may be an alias for components of other variables (the live subset of these variables and `v1` must be annotated with "!" on the assignment statement; checking such annotations is a primary purpose of the sharing analysis). All these variable components must have the same sharing added as `*v1`. The components must be concatenated and folded appropriately. Second, if the assignment does not create a cyclic structure the existing sharing for `v1` can safely be removed, improving precision. It is sufficient to check if any component of `v2` aliases with `v1.[Ref.1]`.

```
alias (DC v dc [v_1, ... v_N]) a0 =          -- v = Dc v1...vN
   let
      self1 = ⋃_{1≤i≤N} ({fc(v.[dc.i]), fc(v.[dc.i])} ∪
                    {{fc(v.(dc.i:c)), fc(v.(dc.i:c))} | {v_i.c, v_i.c} ∈ a0})
      share1 = ⋃_{1≤i≤N} {{fc(v.(dc.i:c_1)), w.c_2} | {v_i.c_1, w.c_2} ∈ a0}
   in
      a0 ∪ self1 ∪ share1
```

The `DerefEq` case can be seen as equivalent to `v1 = Ref v2` and binding a variable to a data constructor with N variable arguments is a generalisation.

```
alias (EqDeref v1 v2) a0 =                  -- v1 = *v2
   let
      self1 = {{v1.c, v1.c} | {v2.(Ref.1 :c), v2.(Ref.1 :c)} ∈ a0}
      share1 = {{v1.c_1, v.c_2} | {v2.(Ref.1 :c_1), v.c_2} ∈ a0}
      empty1 = {{v1.[], v.c} | {v1.[], v.c} ∈ (self1 ∪ share1)}
   in
      if the type of v1 has a [] component then
         a0 ∪ self1 ∪ share1
      else --- avoid bogus sharing with empty component
         (a0 ∪ self1 ∪ share1) \ empty1
```

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.907v1 | CC-BY 4.0 Open Access | rec: 18 Mar 2015, publ: 18 Mar 2015

The `EqDeref` case is similar to the inverse of `DerefEq` in that we are removing `Ref.1` rather than prepending it. However, if the empty component results we must check that such a component exists for the type of `v1`.

```
alias (App v f [v1,...vN]) a0 =            -- v = f v1...vN
   let
      "f(w1,...wK+N) = r" is used to declare sharing for f
      post = the postcondition of f along with the sharing for
               mutable arguments from the precondition,
               with parameters and result renamed with
               f.[Cl.K],...f.[Cl.1], v1,...vN and v, respectively
      postt = {{x1.c1, x3.c3} | {x1.c1, x2.c2} ∈ post ∧ {x2.c2, x3.c3} ∈ a0}
      -- (the renamed precondition of f must be a subset of a0,
      -- and mutable arguments of f and live variables they share
      -- with must be annotated with ! and must not share with
      -- abstract)
      -- selfc+sharec not needed for saturated applications
      selfc = {{v.[Cl.i], v.[Cl.i]} | 1 ≤ i ≤ N} ∪
              {{v.((Cl.i) :c), v.((Cl.i) :c)} |
                  1 ≤ i ≤ N ∧ {vi.c, vi.c)} ∈ a0} ∪
              {{v.(Cl.(i+N)) :c), v.(Cl.(i+N)) :c)} |
                  {f.(Cl.i) :c), f.(Cl.i) :c)} ∈ a0}
      sharec = {{v.(Cl.i) :c1), x.c2} |
                  1 ≤ i ≤ N ∧ {vi.c1, x.c2)} ∈ a0} ∪
              {{v.(Cl.(i+N)) :c1), x.c2} |
                  {f.(Cl.i) :c1), x.c2} ∈ a0}
   in
      a0 ∪ postt ∪ selfc ∪ sharec
```

Function application relies on the sharing information attached to all arrow types. Because Pawns uses the syntax of statements to express pre- and post-conditions, our the implementation uses the sharing analysis algorithm to derive an explicit alias set representation (currently this is done recursively, with the level of recursion limited by the fact than pre- and post-conditions must not contain function calls). Here we ignore the details of how the alias set representation is obtained. The compiler also uses the sharing information to check that preconditions are satisfied, all required "!" annotations are present and abstract variables are not modified.

The main thing done for function application is to add the declared post-condition of the function, renamed appropriately. The N arguments of the call replace the last N formal parameters and `v` replaces the formal result. The first K formal parameters represent closure arguments of `f`, so those variables are replaced with `f` and the components are prefixed with the prepresentation for a closure argument. As well as the declared postcondition, sharing for the mutable arguments of the precondition must be included. The analysis of a function definition guarantees than the union of pre- and post-conditions are satisfied when the function returns (assuming the precondition is satisfied initially), but execution cannot add sharing between non-mutable arguments so it is not added here. Thus by not including preconditions in the declared postconditions, precision is improved. It is also necessary to include one step of transisitivity in the sharing information: if the renamed postcondition introduces sharing between variable components $x_1.c_1$ and $x_2.c_2$ and before the function call $x_2.c_2$ shared with $x_3.c_3$ we add sharing between $x_1.c_1$ and $x_3.c_3$.

For some calls we can know statically than a closure cannot result, but in general we must assume that a closure is created and the first N closure arguments share with the N arguments of the function call and any closure arguments of `f` share with additional closure arguments of the result (this requires renumbering of these arguments).

```
alias Error a0 = ∅                              -- error
alias (Case v [(p_1, s_1), ... (p_N, s_N)]) a0 =   -- case v of ...
   let
      old = {{v.c_1, v_2.c_2} | {v.c_1, v_2.c_2} ∈ a0}
   in
      ⋃_{1≤i≤N} aliasCase a0 old v p_i s_i


aliasCase a0 av v (Pat dc [v_1, ... v_N]) s = -- (Dc *v1...*vN) -> s
   let
      avdc = {{fc(v.(dc.i :c_1)), w.c_2} | {fc(v.(dc.i :c_1)), w.c_2} ∈ av}
      rself = {{v_i.[Ref.1], v_i.[Ref.1]} | 1 ≤ i ≤ N}
      vishare = {{fc(v_i.(Ref.1 :c_1)), fc(v_j.(Ref.1 :c_2))} |
                  {fc(v.(dc.i :c_1)), fc(v.(dc.j :c_2))} ∈ av}
      share = {{fc(v_i.(Ref.1 :c_1)), w.c_2} | {fc(v.(dc.i :c_1)), w.c_2)} ∈ av}
   in
      alias s (rself ∪ vishare ∪ share ∪ (a0 \ av) ∪ avdc)
```

For a case expression we return the union of the alias sets obtained for each of the different branches. For each branch we only keep sharing information for the variable we are switching on which is compatible with the data constructor in that branch (we remove all the old sharing, `av`, and add the compatible sharing, `avdc`). Note we use a high level declarative definition for `avdc` (and other variables) which implicitly uses the inverse of `fc`. To deal with individual data constructors we consider pairs of components of arguments i and j which may alias in order to compute possible sharing between v_i and v_j, including self-sharing when $i = j$. The corresponding component of v_i (prepended with `Ref` and folded) may alias the component of v_j. For example, if `v` of type `RTrees` is matched with `Cons *v1 *v2` and `v.[]` self-shares, we need to find the components which fold to `v.[]` (`v.[Cons.2]` and `v.[Cons.1,RNode.2]`) in order to compute the sharing for `v2` and `v1`. Thus we compute that `fc(v2.[Ref.1,Cons.2])` = `v2.[Ref.1]` may alias `v1.[Ref.1,Cons.1,RNode.2]`, which can occur if the data structure is cyclic. The `DC` case cannot introduce cycles as the variable on the left is distinct from the variables in the right but `Assign` can introduce cycles.

```
alias (Instype v1 v2) a0 =                -- v1 = v2::t
   alias (EqVar v1 v2) a0
   -- (if any sharing is introduced between v1 and v2,
   -- v2 must not be indirectly updated later while live)
```

Type instantiation is dealt with in the same way as variable equality, with the additional check that if any sharing is introduced, the variable with the more general type is not implicitly updated later while still live (it is sufficient to check there is no "`!v2`" annotation attached to a later statement).

6 Example

We now show how this sharing analysis algorithm is applied to the binary search tree code given earlier. We give a core Pawns version of each function and the alias set before and after each statement, plus an additional set at the end which is the union of the pre- and post-conditions of the function. To save space, we write the alias set as a set of sets where each inner set represents all sets containing exactly two of its members. Thus $\{\{a, b, c\}\}$ represents a set of six sharing pairs: sharing between all pairs of elements, including self-sharing. The return value is given by variable `ret` and variables `absL` and `absT` are the versions of `abstract` for type `Ints` and `Tree`, respectively.

```
list_bst xs =                     -- 0
    v1 = Empty                    -- 1
    *tp = v1                      -- 2
    list_bst_du xs !tp            -- 3
    ret = *tp                     -- 4
```

We start with the precondition: $a_0 = \{\{$`xs.[Cons.1], absL.[Cons.1]`$\}$, $\{$`xs.[], absL.[]`$\}\}$. Binding to a constant introduces no sharing so $a_1 = a_0$. $a_2 = a_1 \cup \{$`tp.[Ref.1]`$\}$. The function call has precondition $a_0 \cup \{\{$`tp.[Ref.1]`$\}$, $\{$`tp.[Ref.1,Node.2]`$\}\}$, which is a superset of a_2. Since `tp` is a mutable argument the precondition sharing for `tp` is added: $a_3 = a_2 \cup \{\{$`tp.[Ref.1, Node.2]`$\}\}$. The final sharing includes the return variable, `ret`: $a_4 = a_3 \cup \{\{$`ret.[],tp.[Ref.1]`$\}$, $\{$`ret.[Node.2],tp.[Ref.1,Node.2]`$\}\}$. After removing sharing for the dead (local) variable `tp` we obtain a subset of the union of the pre- and post-conditions, which is $a_0 \cup \{\{$`ret.[],absT.[]`$\}$, $\{$`ret.[Node.2], absT.[Node.2]`$\}\}$.

```
list_bst_du xs !tp =                -- 0
    case xs of
    (Cons *v1 *v2) ->               -- 1
       x = *v1                      -- 2
       xs1 = *v2                    -- 3
       v3 = bst_insert_du x !tp     -- 4
       v4 = list_bst_du xs1 !tp     -- 5
       ret = v4                     -- 6
    Nil ->                          -- 7
       ret = ()                     -- 8
    -- after case                   -- 9
```

We start with the precondition, $a_0 = \{\{\texttt{tp.[Ref.1]}\}, \{\texttt{tp.[Ref.1,Node.2]}\}, \{\texttt{xs.[Cons.1],absL.[Cons.1]}\}, \{\texttt{xs.[],absL.[]}\}\}$. The `Cons` branch of the case introduces sharing for `v1` and `v2`: $a_1 = a_0 \cup \{\{\texttt{xs.[Cons.1]}, \texttt{absL.[Cons.1]}, \texttt{v1.[Ref.1]}, \texttt{v2.[Ref.1,Cons.1]}\}, \{\texttt{v2.[Ref.1]}, \texttt{xs.[]}, \texttt{absL.[]}\}\}$. The list elements are atomic so $a_2 = a_1$. The next binding makes the sharing of `xs1` and `xs` the same: $a_3 = a_2 \cup \{\{\texttt{v2.[Ref.1]}, \texttt{xs.[]}, \texttt{xs1.[]}, \texttt{absL.[]}\}, \{\texttt{v1.[Ref.1]}, \texttt{xs.[Cons.1]}, \texttt{xs1.[Cons.1]}, \texttt{absL.[Cons.1]}, \texttt{v2.[Ref.1,Cons.1]}\}\}$. This can be simplified by removing the dead variables `v1` and `v2`. The precondition of the calls are satisfied and $a_6 = a_5 = a_4 = a_3$. For the `Nil` branch we remove the incompatible sharing for `xs` from a_0: $a_7 = \{\{\texttt{tp.[Ref.1]}\}, \{\texttt{tp.[Ref.1,Node.2]}\}, \{\texttt{absL.[Cons.1]}\}, \{\texttt{absL.[]}\}\}$ and $a_8 = a_7$. Finally, $a_9 = a_6 \cup a_8$. Ignoring local variables, this is a subset of the union of the pre- and post-conditions, a_0.

```
bst_insert_du x !tp =                          -- 0
    v1 = *tp                                   -- 1
    case v1 of
    Empty ->                                   -- 2
        v2 = Empty                             -- 3
        v3 = Empty                             -- 4
        v4 = Node v2 x v3                      -- 5
        *!tp := v4                             -- 6
        ret = ()                               -- 7
    (Node *lp *v5 *rp) ->                      -- 8
        n = *v5                                -- 9
        v6 = (x <= n)                          -- 10
        case v6 of
        True ->                                -- 11
            v7 = (bst_insert_du x !lp) !tp     -- 12
            ret = v7                           -- 13
        False ->                               -- 14
            v8 = (bst_insert_du x !rp) !tp     -- 15
            ret = v8                           -- 16
        -- end case                            -- 17
    -- end case                                -- 18
```

Here $a_0 = \{\{$`tp.[Ref.1]`$\}, \{$`tp.[Ref.1,Node.2]`$\}\}$ and $a_1 = a_0 \cup \{\{$`v1.[]`, `tp.[Ref.1]`$\}, \{$`tp.[Ref.1,Node.2]`, `v1.[Node.2]`$\}\}$. For the `Empty` branch we remove the `v1` sharing so $a_4 = a_3 = a_2 = a_0$ and $a_5 = a_4 \cup \{\{$`v4.[]`$\}, \{$`v4.[Node.2]`$\}\}$. After the destructive update, $a_6 = a_5 \cup \{\{$`v4.[]`, `tp.[Ref.1]`$\}, \{$`v4.[Node.2]`, `tp.[Ref.1,Node.2]`$\}\}$ (`v4` is dead and can be removed) and $a_7 = a_6$. For the `Node` branch we have $a_8 = a_1 \cup \{\{$`v1.[]`, `tp.[Ref.1]`, `lp.[Ref.1]`, `rp.[Ref.1]`$\}, \{$`tp.[Ref.1,Node.2]`, `lp.[Ref.1,Node.2]`, `rp.[Ref.1,Node.2]`, `v5.[Ref.1]`, `v1.[Node.2]`$\}\}$. The same set is retained for $a_9 \ldots a_{17}$ (assuming the dead variable `v5` is retained), the preconditions of the function calls are satisfied and the required annotations are present. Finally, $a_{18} = a_{17} \cup a_7$ and after eliminating local variables we get the postcondition, which is the same as the precondition.

7 Discussion

It is inevitable we lose some precision with recursion in types. However, it seems that some loss of precision could be avoided relatively easily. The use of the empty path to represent sub-components of recursive types results in imprecision when references are created. For example, the analysis of `*vp = Nil; v = *vp` concludes that the empty component of `v` may share with itself and the `Ref` component of `vp` (in reality, `v` has no sharing). Instead of the empty path, a dummy path of length one could be used. A more agresive approach would be to unfold the recursion an extra level, at least for some types. This could allow us to express (non-)sharing of separate subtrees and whether data structures are cyclic, at the cost of more variable components and more complex pre- and postconditions.

Increasing the number of variable components also affects efficiency. The algorithmic complexity is affected by the representation of alias sets. Currently we use a naive implementation, using just ordered pairs of variable components as the set elements and a set library which uses an ordered binary tree. The size of the set can be $O(N^2)$, where N is the maximum number of live variable components of the same type at any program point (each such variable component can share with all the others). In typical code the number of live variables at any point is not particularly large. If the size of alias sets does become problematic, a more refined set representation could be used, such as the set of sets of pairs representation we used in Section 6, where sets of components which all share with each other are optimised. We have not stress tested our implementation as it is intended to be a prototype, but performance has not been concerning at this stage.

8 Related work

Related programming languages are discussed in [1]; here we restrict attention to work related to the sharing analysis algorithm. The most closely related work is that done in the compiler for Mars [6], which extends similar work done for

Mercury [7] and earlier for Prolog [8]. All use a similar abstract domain based on the type folding method first proposed in [5]. Our abstract domain is somewhat more precise due to inclusion of self-aliasing, and we have no sharing for constants. In Mars it is assumed that constants other than numbers can share. Thus for code such as `xs = []; ys = xs` our analysis concludes there is no sharing between `xs` and `ys` whereas the Mars analysis concludes there may be sharing.

One important distinction is that in Pawns sharing (and mutability) is declared in type signatures of functions so the Pawns compiler just has to check the declarations are consistent, rather than infer all sharing from the code. However, it does have the added complication of destructive update. As well as having to deal with the assignment primitive, it complicates handling of function calls and case statements (the latter due to the potential for cyclic structures). Mars, Mercury and Prolog are essentially declarative languages. Although Mars has assignment statements the semantics is that values are copied rather than destructively updated — the variable being assigned is modified but other variables remain unchanged. Sharing analysis is used in these languages to make the implementation more efficient. For example, the Mars compiler can often emit code to destructively update rather than copy a data structure because sharing analysis reveals no other live variables share it. In Mercury and Prolog the analysis can reveal when heap-allocated data is no longer used, so the code can reuse or reclaim it directly instead of invoking a garbage collector.

These sharing inference systems use an explicit graph representation of the sharing behaviour of each segment of code. For example, code s_1 may cause sharing between (a component of) variables `a` and `b` (which is represented as an edge between nodes `a` and `b`) and between `c` and `d` and code s_2 may cause sharing between `b` and `c` and between `d` and `e`. To compute the sharing for the sequence s_1;s_2 they use the "alternating closure" of the sharing for s_1 and s_2, which constructs paths with edges alternating from s_1 and s_2, for example `a-b` (from s_1), `b-c` (from s_2), `c-d` (from s_1) and `d-e` (from s_2).

The sharing behaviour of functions in Pawns is represented explicitly, by a pre- and postcondition and set of mutable arguments but there is no explicit representation for sharing of statements. The (curried) function `alias s` represents the sharing behaviour of `s` and the sharing behaviour of a sequence of statements is represented by the composition of functions. This representation has the advantage that the function can easily use information about the current sharing, including self-sharing, and remove some if appropriate. For example, in the `[]` branch of the case in the code below the sharing for `xs` is removed and we can conclude the returned value does not share with the argument.

```
map_const_1 :: [t] -> [Int]
    sharing map_const_1 xs = ys pre nosharing post nosharing
map_const_1 xs =
    case xs of
        [] -> xs    -- can look like result shares with xs
        (_:xs1) -> 1:(map_const_1 xs1)
```

Given the extra precision that can be achieved, it may be worth attempting to adapt our approach to inferring alias information in languages without destructive update. There are other approches to and uses of alias analysis for imperative languages, such as [9] and [10], but these are not aimed at precisely capturing information about dynamically allocated data. A more detailed discussion of such approaches is given in [6].

9 Conclusion

Purely declarative languages have the advantage of avoiding side effects, such as destructive update of function arguments. This makes it easier to combine program components, but some algorithms are hard to code efficiently without flexible use of destructive update. A function can behave in a purely declarative way if destructive update is allowed, but restricted to data structures which are created inside the function. The Pawns language uses this idea to support flexible destructive update encapsulated in a declarative interface. It is designed to make all side effects "obvious" from the source code. Because there can be sharing between the representations of different arguments of a function, local variables and the value returned, sharing analysis is an essential component of the compiler. It is also used to ensure "preservation" of types in computations. Sharing analysis has been used in other languages to improve efficiency and to give some feedback to programmers but we use it to support important features of the programming language.

The algorithm operates on (heap allocated) algebraic data types, including arrays and closures. In common with other sharing analysis used in declarative languages it supports binding of variables, construction and deconstruction (combined with selection or "case") and function/procedure calls. In addition, it supports explicit pointers, destructive update via pointers, creation and application of closures and pre- and post-conditions concerning sharing attached to type signatures of functions. It also uses an abstract domain with additional features to improve precision.

References

1. Naish, L.: An informal introduction to Pawns: a declarative/imperative language. http://people.eng.unimelb.edu.au/lee/papers/pawns (2015) Accessed 2015-03-16.
2. Milner, R., Tofte, M., Macqueen, D.: The Definition of Standard ML. MIT Press, Cambridge, MA, USA (1997)
3. Jones, S.P., Hughes, J., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze12, R., Hudak, P., et al.: Report on the programming language Haskell 98, a non-strict purely functional language, February 1999. (1999)
4. Lippmeier, B.: Type Inference and Optimisation for an Impure World. PhD thesis, Australian National University (June 2009)
5. Bruynooghe, M.: Compile time garbage collection or how to transform programs in an assignment free languages into code with assignments. IFIP TC2/WG2.1 Working Conference on Program Specification and Transformation, Bad-Tölz, Germany, April 15-17, 1986 (1986)

6. Giuca, M.: Mars: An imperative/declarative higher-order programming language with automatic destructive update. PhD thesis, University of Melbourne (2014)
7. 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
8. Mulkers, A.: Live data structures in logic programs, derivation by means of abstract interpretation. Springer-Verlag (1993)
9. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural aliasing. SIGPLAN Not. **27**(7) (July 1992) 235–248
10. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. PLDI '94, New York, NY, USA, ACM (1994) 242–256