Towards an Effects System for Ownership Domains[⋆]

Matthew Smith

Imperial College London

Abstract. Effects systems can capture the parts of the heap affecting or affected by some predicate or execution. Comparison of such effects can demonstrate the 'independence' of expressions and predicates thus allowing expressions to be safely re-ordered or assuring the preservation of predicates by execution.
We develop an effects system for Aldrich and Chambers' ownership domains based on the Joe system of Clarke and Drossopoulou. We demonstrate our effects through an example, discuss some limitations of the system and suggest extensions.

1 Introduction

Effects systems [7, 8, 11, 5] characterise execution of programs in terms of the part of the heap that is read or written. Additionally, in previous work[18] we extended calculation of effect to predicates with the effect describing the part of the heap needed to determine satisfaction.

Effects describe heaps in terms of an abstraction based on a feature or property of the language. In the language Joe[5] ownership types are exploited to describe effects. We employ ownership domains [1], a recent development of ownership types, to calculate effects which are more precise than those of Joe. We base our system on Joe and extend it to exploit the features of ownership domains.

Effects are useful for determining *independence* properties[1]. In this work independence of expressions means that the expressions can be reordered or parallelised without changing the outcome. We extend the notion to predicates where an expression and predicate are independent if execution of the expression does not affect satisfaction of the predicate. We write $e\#e'$ to mean that expressions e and e' are independent, and $e\#P$ meaning predicate P and expression e are independent (following Reynolds[15, 16] and others).

Independence can be detected by comparing effects of expressions/predicates. A static *disjointness* judgement determines if two effects refer to disjoint parts of the heap. If the read effect of an expression is disjoint from the write effect of

[⋆] Work partly supported by a gift from Microsoft Research

[1] Sometimes called non-interference, which we avoid because of the use of the term in security analysis

another (and vice versa) then the expressions are independent[2]. If the write effect of an expression is disjoint with the effect of a predicate, they are independent.

Independence, via effects has been used to parallelise code [7] and has been shown to allow reordering of expressions [8, 5] e.g. for code transformation. In previous work we suggested applying effects to admit a rule of constancy [16] to a Hoare logic using the effects system of Joe [5]. The rule has the form

$$\frac{\{P\}e\{Q\} \quad R\#e}{\{P \wedge R\}e\{Q \wedge R\}}$$

where independence of R and e is judged using effects.

Ownership domains [1, 10] extend and generalise ownership types giving greater flexibility but still offering strong encapsulation. Because of this we can use ownership domains to calculate accurate effects for a variety of programs. Ownership domains, like ownership types, impose a tree structure on the heap but give more fine grained control by allowing objects to control multiple *domains* of ownership. Additionally, domains give the programmer the power to set the encapuslation policy of the system, providing greater flexibility. Based on the effects systems of Joe[5] our system describes effects based on the tree structure of the heap. The flexibility of ownership domains means that, with our basic system, there are programs for which we cannot calculate accurate effects. We suggest extensions to both ownership domains and our effects to cope with some of these problems.

Section 2 describes ownership domains in more detail. An example follows in Section 3 which introduces our effects and shows independence properties derived from them. Section 4 describes the effects in more detail (though informally due to limitations of space) and Section 5 addresses effects for more complicated programs. We discuss related work and conclude in sections 6 and 7.

2 Ownership Domains

Ownership Domains [1, 10] are an ownership type system which provides a more general and flexible form of ownership than previous systems. Ownership domains have been presented in a general form [10] but we work in the context of the original Java-like setting [1].

Every object is contained within a unique *domain*. Domains are declared in classes and for each object instance of a class there are instances of each declared domain, immutably tied to that object. Domains belonging to the same object are disjoint i.e. they do not share any objects. References between domains are constrained by explicit, programmer granted permissions and a small number of general rules.

In ownership type systems the rules governing which references (i.e. fields) are permitted follow the 'owners as dominators' principle. An object may only reference objects it (directly) owns, its (transitive) owners and objects directly

[2] the read effect always includes the write effect

owned by a transitive owner. Ownership domains are similar in that objects are encapsulated within a tree structure but they have no fixed policy restricting aliases. The programmer may grant reference permissions between domains more or less as they see fit. By this mechanism strong restrictions such as those of ownership types can be encoded but more liberal policies can also be employed.

In Figure 1 we show a class, `A` and, through a diagram, a possible instance of it. In the diagram open arrows represent references (fields) and closed arrows permissions between domains. In class `A` the parameters to the class name i.e. `owner` and `d`, are ownership domains (indicated in the diagram with square cornered boxes, objects having round corners). The special parameter `owner` indicates the domain containing an instance of the class. `A` declares two domains, `a` and `b`, shown by the boxes adjoined to the `A` object.

The types of fields may be instantiated with any domains in scope i.e. parameters and locally declared domains (prefixed with `this.`). The domains used to instantiate a type must obey the assumptions declared in the class. Assumptions are of the form d`->`d' meaning objects in domain d have permission to refer to objects in domain d'. Domains may be explicitly linked together (granting permission for one to access the other) as in `link b -> d`.

The assumed link `owner -> d` allows objects in the `owner` domain to reference objects in the `d` domain, this permits the field `f`. Similarly, `link b -> d` permits objects in `b` to access objects in `d`. The fields `g` and `h` are allowed as an object can always refer to objects in its declared domains. Domain declarations may be marked `public`, in which case they may be referenced by any object which may access the declaring object without the need for an explicit link. The domains `a` and `b` may not refer to each other unless explicitly linked.

```
class A<owner, d>
      extends Object<owner>
          assume owner -> d {
   domain a,b;
   link b -> d;
   Object <d> f;
   Object <this.a> g;
   Object <this.b> h;
}
```

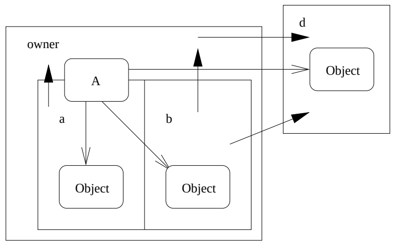

Fig. 1. A class with ownership domains and a possible instantiation

3 Example

Figure 2 gives code, in a Java-like language with ownership domains, for a simple project timetabling programming.

```
class Timetable<owner, projects>
        assume owner -> projects {
    domain durs;
    link owner -> durs;
    Duration<durs> d;
    Project<projects> p;
    Timetable<owner, projects> next;
    int delay(int x){
        this.d.delay(x);
        if(this.next!=null){
            this.next.delay(x);
        }
    }
    ...
}

class Duration<owner> {
    int start;
    int dur;
    void delay(int x) {
        this.start = this.start+x;
    }
    ...
}
```

```
class Project<owner> {
    public domain durs;
    Duration<durs> d;
    ...
}

class Employee<owner,projects>
        assumes owner -> projects {
    domain official, unofficial;
    link official -> projects;
    link unofficial -> projects
    final Timetable<official, projects> o;
    final Timetable<unofficial, projects> u;
    void delayo(int x) {
        this.o.delay(x);
    }
    void delayu(int x) {
        this.u.delay(x);
    }
    ...
}
```

Fig. 2. Project management with ownership domains

`Employee` objects keep a record of the projects they are working on in `Timetable` objects which are contained in the local domains `official` and `unofficial`. A `Timetable` is a sequence of pairs of `Project` and `Duration` objects, indicating the project to be worked on and the time period for the work. Each `Timetable` object declares a domain `durs` which contains the relevant `Duration` object. `Project` objects are stored in the parameter domain `project` thus allowing multiple timetables/employees to refer to the same project.

Figure 3 shows a possible instantiation of the classes from Fig. 2. Each `Timetable` sequence is completely contained within a domain, and their `Duration` objects also (transitively). Projects can be shared between employees as we expect but timetables cannot.

We also assume a predicate `nonOverlapping` which takes a `Timetable` as an argument and is satisfied when the durations of no two elements of the `Timetable` overlap. We omit the definition of the predicate for brevity's sake, it could easily be encoded in a simple predicate language with recursion.

3.1 Effects

In Figure 4 we give effects for each of the methods described above. Effects for expressions (and methods) have the form `rd` ϕ `wr` ϕ' where ϕ is the read effect (the part of the heap read, via field accesses) and ϕ' is the write effect (the part of the heap modified, via field assignments). The read part of the effect always contains the write part.

The `delay` method of `Duration` has effect `rd this wr this` meaning that only the receiver of a call to `delay` is inspected/modified by the call.

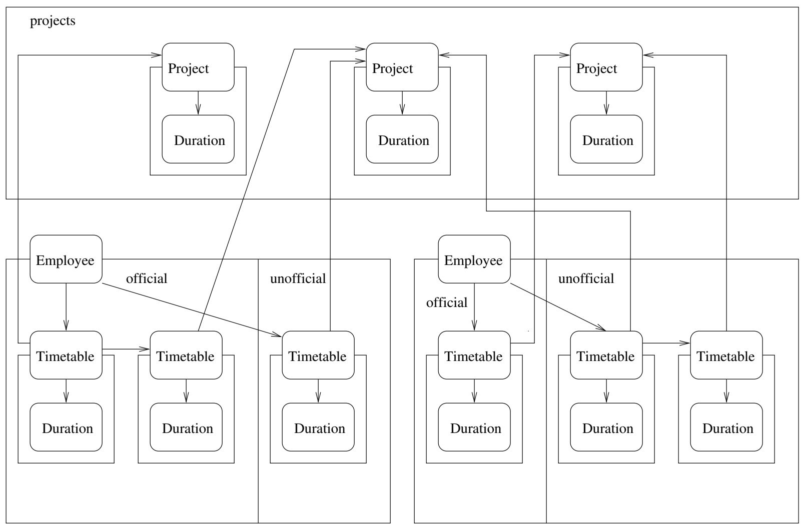

Fig. 3. Instance of classes from Figure 2

The effect of `delay` in the `Timetable` class is more complicated. The method may make a recursive call to the `delay` method of the next `Timetable` in the sequence. The declared effect must describe reads/writes to the current receiver (and the `Duration` object which is delayed) as well as the equivalent effect on the rest of the sequence. The effect `rd owner.under wr owner.under` achieves this. `owner.under` means all the objects in domain `owner` and all those transitively contained i.e. in domains nested inside `owner`. This includes all the `Timetable` objects in the sequence (since they all have the same owner) as well as the `Duration` objects which are nested inside.

The effect declarations for the two delay methods in `Employee` must include the effects of the `Timetable delay` call they make and add the effect of reading their own field. For example the effect of `delayo` is `rd this+this.official.under wr this.official.under`; `this.official.under` is equivalent to `owner.under` in `Timetable` (as the

Class	Method	Effect
`Duration<owner>`	`delay(int x)`	`rd this wr this`

`Timetable<owner, projects>`	`delay(int x)`	`rd owner.under`
		`wr owner.under`

`Employee<owner, projects>`	`delayo(int x)`	`rd this+this.official.under`
		`wr this.official.under`

`Employee<owner, projects>`	`delayu(int x)`	`rd this+this.unofficial.under`
		`wr this.unofficial.under`

Fig. 4. Effects for methods

field `o` has type `Timetable<official, projects>`) and `this` records the read of field `this.o`.

As in our previous work [18] we can calculate an effect for the predicate `nonOverlapping` in a similar way to calculating the effect of an expression. We give this effect in Figure 5. Predicate effects have the form ϕ as they only have a 'read' part. The predicate needs to compare all the `Duration` objects of each `Timetable` object in the sequence beginning at `t`, so must read fields of the `Timetable` objects and the `Duration` objects. The effect `owner.under` includes all the `Timetable` and `Duration` objects in the sequence and so is an appropriate effect for `nonOverlapping`.

Predicate	Effect
`nonO(Timetable<owner,projects> t)`	`owner.under`

Fig. 5. Effects for methods and predicates

3.2 Disjointness and Independence

By comparing the effects of expressions and predicates for the above program we can find pairs of expressions and predicates to be independent. Throughout we assume that we are working in the context of an environment where `a` and `b` are final variables of type `Employee<owner,projects>`. That is, they are each owned by the same domain and their timetables point to projects in the same domain. We assume further that they are known not to be aliases.

`a.delayo(5)#b.delayo(23)` The expressions `a.delayo(5)` and `b.delayo(23)` have effects as follows (based on the types of the variables in our assumed environment and the effects we have given for the methods):

```
 a.delayo(5):rd a+a.official.under wr a.official.under
b.delayo(23):rd b+b.official.under wr b.official.under
```

Recall that two expressions are independent if the read effect of each expression is disjoint from the write effect of the other. Since we know `a` and `b` to not be aliases we know that the domains `a.official` and `b.official` are distinct, they cannot be nested, since `a` and `b` have the same owner. Thus `a.official.under` and `b.official.under` must also be disjoint. Further still `a` cannot be in `b.official.under` since `a` and `b` have the same owner and all objects in `b.official.under` are nested within `owner`. By distributivity since both `a` and `a.official.under` are disjoint from `b.official.under` then `a+a.official.under` is also disjoint from `b.official.under`. Thus `a.delayo(5)` and `b.delayo(23)` are independent. In fact, any combination of delays on `a` and `b` will be disjoint as all the timetables of `a` and `b` are separate.

`a.delayo(5)#a.delayu(17)` The effects are as follows:

```
 a.delayo(5):rd a+a.official.under wr a.official.under
a.delayu(17):rd a+a.unofficial.under wr a.unofficial.under
```

These expressions are also independent. The argument is largely the same as above except that `a.official.under` and `a.unofficial.under` are disjoint because `a.official` and `a.unofficial` are distinct domains (different names) and are not nested (they belong to the same object).

`nonOverlapping(a.o)#a.delayu(17)` The predicate `nonOverlapping(a.o)` and `a.delayu(17)` are independent. This is because the write effect of `a.delayu(17)`, `a.unofficial.under` is disjoint from the effect of `nonOverlapping(a.o)`, `a.official.under`. These effects are disjoint by the same argument as in Sec. 3.2. The independence of `nonOverlapping(a.o)` `a.delayu(17)` with the rule of constancy and an appropriate Hoare logic would allow the following deduction:

$$\frac{\{P\}\texttt{a.delayu(17)}\{Q\} \quad \texttt{nonOverlapping(a.o)\#a.delayu(17)}}{\{P \wedge \texttt{nonOverlapping(a.o)}\}\texttt{a.delayu(17)}\{Q \wedge \texttt{nonOverlapping(a.o)}\}}$$

4 Effects

In this section we describe the core of the effects system we propose for ownership domains. We present a grammar for effects and describe the rest of the system informally. The effects describe heap structure in terms of the tree structure of domains (where nesting of domains forms a tree). The grammar of effects, ψ, is:

$$\begin{array}{lcl}
\psi & ::= & \texttt{rd}\,\phi\,\texttt{wr}\,\phi' \\
\phi & ::= & \theta \mid \theta.\texttt{under} \mid \texttt{emp} \mid \phi + \phi' \\
\theta & ::= & d \mid path \mid path.d \mid path.\texttt{all} \\
path & ::= & z \mid path.f
\end{array}$$

where z ranges over program variables, f over field identifiers and d over the names of parameter domains and locally declared domains.

The meaning of effects is as follows:

d the set of objects in domain d, a parameter to the current receiver (or the top level domain `world`)
$path$ the object denoted by the sequence of field accesses $path$
$path.d$ the set of objects in the domain d of the object pointed to by $path$
$path.$`all` all objects in all the domains of the $path$ object
$\theta.$`under` all the objects below the object(s) denoted by θ in the tree (including the object(s) in θ)
`emp` the empty shape, denoting no objects
$\phi + \phi'$ the union of ϕ and ϕ'

For matters of soundness, all paths in effects, e.g. $path.d$, must be final i.e. a sequence of final field accesses rooted at a final variable. These terms must be constant throughout execution in order for the effect to be valid. If they were not, the meaning of an effect annotation before and after a computation could be different and the system would not be sound. Similar restrictions are required in both the effects of Joe, where all local variables are final [5] and the original domain system [1] where final paths are required to make type instantiations sound.

4.1 Disjointness of Effects

Judgement of disjointness for our effects is formed from a combination of typing, syntactic comparison and structural properties of trees. We describe a selection of rules from the system.

Pairs of domain parameters cannot be detected disjoint as we have no knowledge of their position in the tree, only of the links between them. Simple paths can be shown disjoint when their types are incompatible i.e. neither is a subtype of the other, since they cannot refer to the same object. Further from this, for any disjoint paths, $path$ and $path'$ any local domains of these paths, $path.d$ and $path'.d'$, must be disjoint since domains belong to only one object and these objects are known to be disjoint. Any two $path.d$ effects where the domain names (d) are distinct must be disjoint simply since they have different names.

Structural shapes i.e. $\theta.\texttt{under}$ are more difficult to judge disjoint, as we must be certain that neither shape is nested inside the other. The effects $path.d.\texttt{under}$ and $path.d'.\texttt{under}$ are disjoint when $d \neq d'$ since $path.d$ and $path.d'$ are different domains and thus the sets of all objects each (transitively) contains are disjoint. Because d and d' belong to the same object, $path$ they cannot be nested.

General rules such as distributivity where ϕ is disjoint from $\phi' + \phi''$ if it is disjoint from ϕ' and from ϕ'' and rules allowing subsumption of effects complete the system.

4.2 Joe

Our effects system is based on that of Joe [5]. Both systems exploit the underlying tree structure of ownership types/domains. The grammar for Joe effects is as follows:

$$\begin{array}{lll} \psi & ::= & \texttt{rd}\ \phi\ \texttt{wr}\ \phi \\ \phi & ::= & \emptyset \ \mid\ p.n \ \mid\ \texttt{under}(p.n) \ \mid\ \phi \cup \phi \end{array}$$

In the effects of Joe the ‘root’ of each effect (p in $p.n$) is a local variable or type parameter. The equivalent ‘roots’ in our system are richer as we allow final paths, (rather than just variables) and domain expressions (d, $path.d$, $path.\texttt{all}$) (rather than just parameter domains). A subtle but important distinction is that, in Joe, p whether a variable or a parameter refers to an object (which owns other objects). In our system, because domains and objects are distinct there is a semantic difference between a *path* which refers to a single object and a domain expression (*d* or *path.d*, for example) which refers to a set of objects. In both systems an inclusion relation is used in the effects system to reason when a root is higher in the tree than another. In Joe this is a relation on pairs of objects but in our system it is a relation between pairs of objects (paths), pairs of domains and between domains and objects.

We use the notion of 'under' from Joe (where it is written `under(`*target*`)`) to describe all objects which are under *target* in the ownership tree. Joe includes a more fine grained effect called bands ($p.n$), which describes strata within the tree e.g. `z.1` is the objects owned by `z` whereas `z.2` is the objects owned by those in `z.1`. Such effects could be included for ownership domains but we omit them here for brevity and as they do not add anything to our examples. Selection of all local domains e.g. *path*.`all` is equivalent to the Joe effect $z.1$.

Naturally some of our judgements of effect disjointness etc. follow from those seen in Joe. We have rules equivalent to each of those in Joe as well as additional rules for dealing with features new to our contribution.

5 Further Issues

So far we have discussed ownership domains in general and have presented a simple example. Whilst we have not excluded any part of the system we have not seen the full power of ownership domains. Ownership domains can provide good solutions to programming problems that have proved hard in other ownership type systems. Unfortunately the same features that admit these solutions also make it hard to give accurate effects to methods/expressions in these programs.

In this section we present an example for which we cannot calculate useful effects. We go on to suggest an extension to our effects and to ownership domains which addresses this deficiency.

5.1 Iterators

In [1] an interesting solution is given to a common problem with ownership type systems. Whilst linked lists are the paradigmatic example of the power of ownership types, iterators over such lists are problematic. Ownership types allow the internal structure i.e. link nodes to be encapsulated by making them owned by the list object. This ensures that i) no two lists can share a representation (though they may contain the same data) ii) the list object controls all access to the list representation. Unfortunately, in general, iterators [6] require access to the internal representation of the list they iterate over. Various solutions have been proposed to this problem with varying degrees of success[3].

[3] Clarke and Drossopoulou [5] allow stack variables to break ownership boundaries, allowing an iterator to be created and used within a method body. This has limited usefulness since the reference to the iterator cannot be stored (in a field) by an external client. Boyapati et al. [3] follow a suggestion of Clarke [4] and allow inner classes to refer to owned objects of the enclosing instance but for instances of such inner classes to be passed out to clients.

Figure 6 shows the ownership domains solution to list iterators as presented in [1]. In this solution each `List` has a domain for holding the list representation (`list`) and one for iterators (`iters`). The `iters` domain is public so that clients can refer to iterators of the list. The `getIterator` method of `List` creates a specialised `ListIterator` but it is returned with only the generic `Iterator` interface type. The `Iterator` interface takes only two domain parameters, one for the owner and one for the domain where the list data is stored. The specialised `ListIterator` takes a further parameter, `list`, the domain containing the representation of the list it iterates.

```
class List<owner, elems>
      assumes owner->elems {
   domain list;
   public domain iters;
   link list->elems, iters->elems,
      iters->list;
   Cons<list, elems> head;
   void add(Object<elems> o){...}
   Iterator<iters, elems> getIter(){
      return new ListIterator<iters,
           elems, list>(head);
   }
}

class Cons<owner, e> assumes owner->e {
   Object<e> obj;
   Cons<owner,e> next;
}

interface Iterator<owner, elems> assumes
     owner->elems {
   Object<elems> next();
   boolean hasNext();
}
```

```
class ListIterator<owner, elems, list>
      implements Iterator<owner, elems>
      assumes owner->elems, owner->list
           , list->elems {
   Cons<list, elems> current;
   boolean hasNext(){...}
   Object<elems> next() {
      Object<elems> obj = current.obj;
      current = current.next;
      return obj;
   }
}

class Client<owner> {
   domain d;
   final List<d,d> l = new List<d,d>();
   void run(){
      Object<d> obj = ...
      l.add(obj);
      Iterator<l.iters, d> i = l.
           getIters();
   }
}
```

Fig. 6. Lists and Iterators in Ownership Domains

5.2 Effects for Iterators

In Figure 7 we attempt to give effects for the methods in Fig. 6. Whilst we can easily calculate an effect for `next` in `ListIterator` we cannot calculate a satisfactory effect for `next` in `Iterator`.

The effect of `next` in `ListIterator` should be obvious. The read effect must include `list` as it reads the `next` field of `current`, which is owned by `list`, also it reads and writes the receiver's field `current` so the effect must include `this`.

The declared effect of an overriding/implementing method must be a sub-effect of the declaration in the superclass (c.f. covariant changes to return types). Thus, the effect of `next` in `Iterator` must include `list`. This poses a problem

Class	Method	Effect
`Iterator<owner,elems>`	`next()`	`rd this+? wr this`
`ListIterator<owner,elems,list>`	`next()`	`rd this+list wr this`

Fig. 7. Partial effects for Iterator methods

since in the `Iterator` interface there is no way to mention `list` (since it is not a parameter to the type and there is no structural way to refer to it).

The trivial effect `world.under` would be a sound effect (since it describes the entire heap) but it is also useless as it is only disjoint with the empty effect.

5.3 Constraints and More Expressive Effects

To overcome our problem in calculating effects for Iterators we propose additional constructs to our effects and analogous constraints which will be added to the assumes clause of classes.

We propose the addition of a term `.siblings` to our effects, so that:

$$\phi ::= \ \ldots \mid d.\texttt{siblings} \mid path.d.\texttt{siblings}$$

The meaning of these effects is intended to be all domains belonging to the same object as *d* or *path.d*. Figure 8 shows this graphically where *p* is a path referring to the indicated object and *dom* is an alias for the indicated domain (if *dom* were a parameter say). The effects *p.a*.`siblings` and *dom*.`siblings` are equal and refer to the three shaded objects, that is all the objects in all the domains belonging to *p*. The observant reader may notice that *path.d*.`siblings` is equivalent to *path*.`all`. The effect is still necessary however as it may occur through instantiation of a type, it could be converted to *path*.`all` through subsumption or equivalence.

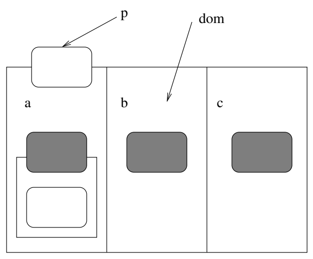

Fig. 8. The sibling effect

Now we can give an effect to both `next` methods as shown in Figure 9. This would appear to be sufficient but in fact we need a little more. Whilst the effect we have given is appropriate for iterators in the context of Figure 6 we have no way of knowing that, in general, the domains `list` and `owner` of `ListIterator` are siblings. It is, of course, the intent that `ListIterator` objects are only ever created by `List` objects, and that `owner` and `list` will only ever be instantiated with sibling domains but that is neither enforced nor statically detectable.

Class	Method	Effect
`Iterator<owner,elems>`	`next()`	`rd this+owner.siblings wr this`
`ListIterator<owner,elems,list>`	`next()`	`rd this+owner.siblings wr this`

Fig. 9. Full effects for Iterator methods

To cope with this we suggest the introduction of further constraints between domains in the `assumes` clause of a class. A constraint of the form d `sibling` d', meaning that domain d is a sibling domain of d', would be checked when the type is instantiated (just like the link assumptions). Adding `owner sibling list` to the assumes clause of `ListIterator` guarantees that `owner` and `list` are always sibling domains and that the effect of `next` is sound.

Statically Checking Siblings Implementing the sibling constraint does not require any fundamental changes to the machinery of ownership domains, only extension. We add rules for checking the constraint in the assumes clause and rules for calculating subshapes involving `siblings`.

Consider class `List` in Figure 6 and assume the addition of `owner sibling list` to the `assumes` clause of `ListIterator`. In the `getIter` method where `ListIterator` is instantiated, the type system will check that `iters` and `list` (the instantiations of `owner` and `list`) are siblings. This is straightforward as `iters` and `list` are both declared in `List`. In general we can deduce that for any *path*, *path.d* `sibling` *path.d'*[4]. Sibling constraints can also be propagated through the `assumes` clause, just as `link` is.

Calling method `next` on `i` in the `run` method of `Client` has the effect `rd i+l.iters.siblings wr i`. By subsumption this becomes `rd i+l.all wr i` (which is, in fact, equivalent) and then the standard rules of our system can be used to reason about the effect.

6 Related Work

Through ownership domains we gain greater expressive power than Joe. In [18] we presented, in Joe, a similar example to that in Figure 2. Were we to reformulate Fig. 2 in Joe each employee would own both timetables but they would not

[4] In the previous sentence `iters` and `list` are shorthands for `this.iters` and `this.list`

be separated as in ownership domains[5]. Using Joe effects we could still deduce the independence described in Sec. 3.2 as each employee would own its timetables and we know the employees to be distinct. Using Joe we could not deduce the independence in Sec. 3.2, however, as the two timetables would be owned by the employee and thus cannot be distinguished (just as would be the case if they were in the same domain).

Ownership domains with our effects are (up to inclusion of bands) demonstrably more powerful than the effects of Joe since i) ownership types can be encoded in ownership domains ii) we have shown properties of independence that could not be deduced in Joe.

The 'regions' effects system of Greenhouse and Boyland [8] and datagroups [11, 12] bear similarities with ownership domains and our effects but have major differences. Both these systems declare groups, not unlike domains, to which fields may belong. The difference is that these regions divide up the fields of an object into sub-objects rather than containing the objects pointed to, as in domains. There is an implicit ownership relation through the use of unique pointers to objects, known as 'pivot' fields. The datagroups of a pivot field may be mapped into those of the owning object allowing the effects of computation on the pivot object to be captured by the datagroups of the owner.

Other systems for object encapsulation could be candidates for development of effects systems based on the structure they enforce on the heap. In practise systems like balloons [2], islands [9] and similar do not seem to offer general enough structure or access to that structure to exploit.

Separation logic [13, 17, 14] seems a rather direct competitor to effects systems and the strong encapsulation provided by ownership domains and similar. Separation logic is able to reason about structures such as linked lists without the need for explicit encapsulation. The 'frame rule' and spatial conjunction are the counterparts to the rule of constancy and static judgement of independence but require the specifier to think 'spatially' as well as logically. Separation logic requires less of the language (no sophisticated type systems as here) and less of the programmer but more of the verifier and specifier.

7 Conclusions and Further Work

We have established the core of an effects system for a language with ownership domains. We have shown, through a simple example, that the effects system we propose is more powerful than the system of Joe on which it is based.

As we have shown in Section 5, we cannot, with our core system give good effects to all programs. We suggested an addition to our effects, which also required an extension to ownership domains. Further work remains to investigate other useful additions to the effects system. Constraints similar to the `sibling` constraint we suggested could be useful in general (not just paired with a related effect). Structural constraints, e.g. constraining a domain to be contained within

[5] In ownership types each object effectively has only one domain which contains all objects it owns.

another would assist in judging disjointness of effects in general. For example constraining a parameter domain to be nested within another would imply the two domains are disjoint. We intend to investigate further. Proof of soundness of the effects system also remains further work.

Acknowledgements

I would like to thank my supervisor, Sophia Drossopoulou for her help and guidance with this work and to the anonymous reviewers for their helpful suggestions. Thanks also go to the members of the SLURP group at Imperial for listening and discussing.

References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP Proceedings*, June 2004.
2. P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP Proceedings*, June 1997.
3. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–223. ACM Press, 2003.
4. D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Australia, 2001.
5. D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 292–310. ACM Press, 2002.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
7. D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon. Fx-87 reference manual. Technical Report 407, M.I.T. Laboratory for Computer Science, September 1987.
8. A. Greenhouse and J. Boyland. An object-oriented effects system. *Lecture Notes in Computer Science*, 1628:205–229, 1999.
9. J. Hogg. Islands: aliasing protection in object-oriented languages. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 271–285. ACM Press, 1991.
10. N. Krishnaswami and J. Aldrich. Permission-based ownership: Encapsulating state in higher-order typed languages. In *Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, New York, NY, USA, 2005. ACM Press.
11. K. R. M. Leino. Data groups: specifying the modification of extended state. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 144–153. ACM Press, 1998.
12. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 246–257. ACM Press, 2002.

13. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*. Springer, 2001.
14. M. Parkinson and G. Bierman. Separation logic and abstraction. In *to be published in Proceedings of POPL*. ACM Press, 2005.
15. J. C. Reynolds. Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 39–46. ACM Press, 1978.
16. J. C. Reynolds. *The Craft of Programming*. Prentice-Hall International Series in Computer Science. Prentice Hall International, 1981.
17. J. C. Reynolds. Separaion logic: A logic for shared mutable data structures. In *Proceedings of LICS*, 2002.
18. M. Smith and S. Drossopolou. Cheaper reasoning with ownership types. In *IWACO Proceedings*, 2003.