Separation Logic and Abstraction

Matthew Parkinson
University of Cambridge
Computer Laboratory
Cambridge CB3 0FD, UK
mjp41@cl.cam.ac.uk

Gavin Bierman
Microsoft Research
7 J J Thomson Ave
Cambridge CB3 0FB, UK
gmb@microsoft.com

ABSTRACT

In this paper we address the problem of writing specifications for programs that use various forms of modularity, including procedures and Java-like classes. We build on the formalism of separation logic and introduce the new notion of an *abstract predicate* and, more generally, abstract predicate families. This provides a flexible mechanism for reasoning about the different forms of abstraction found in modern programming languages, such as abstract datatypes and objects. As well as demonstrating the soundness of our proof system, we illustrate its utility with a series of examples.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification—*class invariants*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, packages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and inheritance*

General Terms

Languages, Theory, Verification

Keywords

Separation Logic, Modularity, Resources, Abstract data types, Classes

1. INTRODUCTION

In order to assist programmers in building complex software systems, programming languages offer various forms of abstraction. In this paper we focus on those that provide some form of modularity. These range from simple procedures with local state, through abstract datatypes (ADTs), to the complexities of Java-like class hierarchies with method overriding and runtime resolution of method invocation.

Our aim is to provide intuitive ways for programmers to specify the behaviour of their modular code. Previous solutions to handling modularity are either too weak, in that certain natural specifications can not be expressed; or too strong, in that the programmer is forced to accept an unreasonable proof or annotation burden.

POPL'05, January 12–14, 2005, Long Beach, California, USA.

We choose to build upon the recent formalism of separation logic, which facilitates local reasoning about code [23]. This local reasoning approach has proved successful when considering many real world algorithms, including the Schorr-Waite graph marking alrithm [31] and a copying garbage collector [4].

Until recently, the work on separation logic has focused exclusively on low-level C-like languages with no support for abstraction. O'Hearn, Reynolds and Yang [22] have recently added *static* modularity to separation logic. They hide the internal resources of a module from its clients using the so called hypothetical frame rule. This partitioning of resources between the client and the module allows them to model "ownership transfer", where state can safely be transferred between the module and the client without fear of dereferencing dangling pointers. This allows them to reason about examples such as a simple memory manager, which allocates fixed size blocks of memory, and a queue.

Though this is a significant advance, their work is severely limited as it only models static modularity. Their modules are based on Parnas' work on information hiding [25], which deals with single instances of the hidden data structure. Hence, it can not be used for many common forms of abstraction, including ADTs and classes, where we require multiple instances of the hidden resource. For example, one would expect, given a list module, to use multiple lists in an application; and one frequently creates new objects in object-oriented applications.

Let us review the problem: take a piece of code that we wish to consider "abstract" (this could be because the code is a procedure, a module or a method). A specification is then a contract between the code and its callers. It includes a precondition that expresses what a caller must establish before the code may be executed. The implementation of the module can assume the precondition on entry. A specification also contains a postcondition that records what must hold upon exit of the module. Consequently the caller can assume the postcondition upon return from the module. When reasoning about the module and the calls, only the contract given by the specification is used: that is, we expect the appropriate form of information hiding.

Various researchers have proposed enriching the logic to view the data abstractly (as in data groups [14]), or the methods/procedures abstractly (as in method groups [30, 13]). In contrast, we propose to add the abstraction to the logical framework itself, by introducing the notion of an *abstract predicate*. An abstract predicate has a name, a definition, and a scope. Within the scope one can freely swap between using the abstract predicate's name and its definition, but outside its scope it must be handled atomically, i.e. by its name. Thus the scope defines the abstraction boundary for the abstract predicate.

In various work on separation logic (e.g. [29]) it is common

to use inductively defined predicates to represent data types. In essence we allow predicates to additionally encapsulate state and not just represent it. This gives us two key advantages: (1) the impact of changing a predicate is easy to define; and (2) by encapsulating state we are able to reason about ownership transfer.

Whilst the notion of abstract predicates is sufficient to reason about modules and simple ADTs, we should like to reason about object-oriented forms of abstractions; more precisely Java-like classes and inheritance. This adds an additional complication: not only do we have to reason about encapsulation but also inheritance. Rather pleasingly this again can be provided by reflecting the abstraction in the logical framework itself. Here the key observation is that an object can exist at multiple types through the class hierarchy. We reflect this in the logic by generalising abstract predicates to *families* of abstract predicates that are indexed by class.

The rest of the paper is structured as follows. In §2 we give a brief overview of separation logic, detailing the features that we use in this paper. In §3 we present more formally the notion of an *abstract predicate*, giving proof rules and outlining a soundness proof. We also give a number of worked examples. In §4 we extend these reasoning principles to a core subset of Java. Again we outline a soundness proof and give examples. We conclude in §5 with a comparison to related work and propose some future work.

2. SEPARATION LOGIC PRIMER

In this section we give some brief details of the fragment of separation logic that we shall use. Space prevents us giving a complete description or explanation of the significant advantages of using separation logic. The interested reader can read further details and references in a survey paper by Reynolds [29].

Separation logic is an extension to Hoare logic that permits reasoning about shared mutable state. It extends Hoare logic by adding spatial connectives to the assertion language, which allow assertions to define separation between parts of the heap. This separation provides the key feature of separation logic—*local reasoning*—specifications need only mention the state they access [23].

We use the standard model of state from separation logic. A heap, *H*, is a partial function from locations to values (for simplicity we take Values to be the integers and Locations to be the positive integers).

$$\mathcal{H} \stackrel{\text{def}}{=} \mathsf{Locations} \rightharpoonup_{fin} \mathsf{Values}$$

This has a partial commutative monoid for disjoint function composition:

$$H_1 * H_2 \stackrel{\text{def}}{=} \lambda l. \begin{cases} H_1(l) & l \in dom(H_1) \\ H_2(l) & l \in dom(H_2) \end{cases}$$

which is defined iff $dom(H_1) \cap dom(H_2) = \emptyset$. A stack, *S*, is a function from (program) variables to values. Unlike other presentations [22], we do not interpret auxiliary variables[1] using the stack but we define an auxiliary stack, *I*, that is a function from auxiliary variable names to values.[2]

$$\mathcal{S} \stackrel{\text{def}}{=} \mathsf{Vars} \to \mathsf{Values}$$
$$\mathcal{I} \stackrel{\text{def}}{=} \mathsf{AuxVars} \to \mathsf{Values}$$

We define a state as a triple consisting of a stack, a heap and an auxiliary stack. A predicate is just a set of states, and formulae are given by the following grammar where B and E range over boolean- and integer-valued expressions respectively (these are defined formally in the §3.1).

[1]Sometimes called ghost or logical variables.

[2]We add this as we make heavy use of local variables, and do not have global variables.

$$\begin{array}{rcl} P, Q & ::= & B \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \\ & \mid & empty \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \mathrm{E} \mapsto \mathrm{E}' \end{array}$$

The usual classical connectives ($\neg, \vee, \wedge, \Rightarrow$) are interpreted using the boolean algebra structure induced on the powerset of states. In addition to the boolean connectives we have the new spatial connectives $*$ and $\mathrel{-\!\!*}$, along with the predicates $empty$ and $\mapsto$. Taking these in reverse order: the predicate $\mathrm{E} \mapsto \mathrm{E}'$ consists of all the triples (S, H, I) where the heap, *H*, consists of the single mapping from the location given by the meaning of E to the value given by the meaning of E'.

$$S, H, I \models \mathrm{E} \mapsto \mathrm{E}' \stackrel{\text{def}}{=} dom(H) = \{[\![\mathrm{E}]\!]_{S,I}\} \ \wedge \ H([\![\mathrm{E}]\!]_{S,I}) = [\![\mathrm{E}']\!]_{S,I}$$

We use the shorthand $E \mapsto E_1, E_2$ to mean $E \mapsto E_1 * E + 1 \mapsto E_2$.

The spatial conjunction $P * Q$ means the heap can be split into two disjoint parts in which P and Q hold respectively.

$$S, H, I \models P * Q \stackrel{\text{def}}{=} \exists H_1, H_2.\, H_1 * H_2 = H \wedge S, H_1, I \models P \wedge \ S, H_2, I \models Q$$

Heaps of more than one element are specified by using the $*$ to join smaller heaps. The $*$ has a unit $empty$ that consists of all states (S, H, I) where H is the empty heap. The adjunct to $*$, written $\mathrel{-\!\!*}$, is not used in this paper so we shall suppress its (routine) definition.

The essence of "local reasoning" is that to understand how a piece of code works it should only be necessary to reason about the memory the code actually accesses (its so-called "footprint"). Ordinarily aliasing precludes such a principle but the separation enforced by the $*$ connective allows this intuition to be captured formally by the following rule.

FRAME RULE

$$\frac{\vdash \{P\}C\{Q\}}{\vdash \{P * R\}C\{Q * R\}}$$

where C does not modify the free variables of R, i.e. $\mathrm{modifies}(C) \cap FV(R) = \emptyset$.

The side-condition is required because $*$ only describes the separation of heap locations and not variables; see [5] for more details. **Note:** modifies(C) denotes the set of stack variables assigned by a given command, C, e.g. $\mathrm{modifies}(x{=}3) = \{x\}$. However assignment through a stack variable to the heap is not counted: $\mathrm{modifies}([x]{=}3) = \emptyset$. See [31] for full definition.

By using this rule, a local specification concerning only the variables and parts of the heap that are used by C can be arbitrarily extended as long as the extension's free variables are not modified by C. Thus, from a local specification we can infer a global specification that is appropriate to the larger footprint of an enclosing program.

3. A LANGUAGE WITH MODULES

In this section we consider reasoning about a simple imperative language with first-order functions/procedures, which is essentially the same as that considered by Reynolds [29]. To simplify the presentation we delay using Java to §4. We introduce our novel concept of an abstract predicate, and state some rules for its use. (These rules are proved sound in §3.4.) We demonstrate the power and elegance of abstract predicates in reasoning about modular code by considering two detailed examples: a connection pool and a memory manager.

3.1 Syntax

The syntax for the programming language considered in this section is given by the grammar in Figure 1. We use x to range over

```
C  :=  let k1 x1 = C1, ..., kn xn = Cn in C
   |   return E | x = k(E) | newvar x;C | x = E
   |   x = [E] | [E] = E | x = cons(E) | dispose(E)
   |   if B then C else C | while B C | C;C
E  :=  x | E + E | E − E | E ∗ E | n | null
B  :=  E == E | E ≤ E | true | false
```

Figure 1: Module language syntax

program variable names, and k ranges over function names. We have a distinguished program variable ret that is not modifiable except with the `return` command. We restrict our consideration to well-formed programs: e.g. a well-formed program only has `return`s as the last command of a function; and defines a function name at most once in a `let`. In the examples of §3.3 we will use syntactic sugar for procedures: procedure definitions are functions that return `null`, and procedure calls are functions calls assigned to an unused variable.

The command `newvar` x`;`C defines a new local variable for the command C, we use a shorthand `newvar` x,...,y`;`C for introducing multiple variables; x = `cons`($\overline{\text{E}}$) allocates $|\overline{\text{E}}|$ consecutive heap locations with the values of $\overline{\text{E}}$. The location E is disposed using `dispose`; updated to E′ with [E] = E′; and stored in x with x = [E].

3.2 Proof rules

For the assertion language we take the language given in §2 and extend it with predicates. Naturally we restrict our consideration to well-formed formulae, and again we elide the obvious definition. We write α to range over predicate names and use a function $arity()$ from predicate names to their arity.

A judgement in our assertion language is written as follows:

$$\Lambda; \Gamma \vdash \{P\}C\{Q\}$$

This is read: the command, C, satisfies the specification $\{P\}$-$\{Q\}$, given the function hypotheses, Γ, and predicate definitions, Λ. The hypotheses and definitions are given by the following grammar:

$$\begin{array}{lcl} \Gamma & := & \epsilon \mid \{P\}k(\overline{x})\{Q\}, \Gamma \\ \Lambda & := & \epsilon \mid \alpha(\overline{x}) \stackrel{\text{def}}{=} P, \Lambda \end{array}$$

However, when it simplifies the presentation, we will treat Λ as a partial function from predicate names to formulae, and Γ as a partial function from function names to specifications. We define $\Lambda(\alpha)[\overline{\text{E}}]$ as $P[\overline{\text{E}}/\overline{x}]$ where Λ contains $\alpha(\overline{x}) \stackrel{\text{def}}{=} P$.

For the hypotheses, Γ, to be well-formed each function, k, can appear at most once; and the specification's free program variables are contained in its arguments and ret. For the predicate definitions, Λ, to be well-formed we require that each predicate, α, is contained at most once; the free variables of the body, P, are contained in the arguments, $\overline{x}$; and P is a positive formula.[3] We will only consider well-formed Γ and Λ.

Intuitively, the predicates are used like abstract data types. Abstract data types have a name, a scope and a concrete representation. Within this scope the name and the representation can be freely exchanged, but outside only the name can be used. Similarly abstract predicates have a name and a formula. The formula is scoped: inside the scope the name and the body can be exchanged, and outside the predicate must be treated atomically. Hence our first rule:

ABSTRACT FUNCTION DEFINITION

$$\frac{\begin{array}{c}\Lambda, \Lambda'; \Gamma \vdash \{P_1\}C_1\{Q_1\} \\ \vdots \\ \Lambda, \Lambda'; \Gamma \vdash \{P_n\}C_n\{Q_n\} \\ \Lambda; \Gamma, \{P_1\}k_1(\overline{x_1})\{Q_1\}, \ldots \{P_n\}k_n(\overline{x_1})\{Q_n\} \vdash \{P\}C\{Q\}\end{array}}{\Lambda; \Gamma \vdash \{P\}\texttt{let}\ k_1\ \overline{x_1} = C_1, \ldots, k_n\ \overline{x_n} = C_n\ \texttt{in}\ C\{Q\}}$$

where
- P, Q, Γ and Λ do not contain the predicate names in $dom(\Lambda')$;
- $dom(\Lambda)$ and $dom(\Lambda')$ are disjoint; and
- the functions only modify local variables: $\text{modifies}(C_i) = \emptyset (1 \leq i \leq n)$.

This rule allows a module writer to use the definition of an abstract predicate, yet the client can only use the abstract predicate name. The functions $k_1, \ldots, k_n$ are within the scope of the predicates defined in Λ' hence verifying the function bodies $C_1 ... C_n$ can use the predicate definitions. The client code, C, is *not* in the scope of the predicates, so it can only use the predicates atomically and through the specifications of $k_1, ..., k_n$. The predicate names can not occur in the conclusions specification, P and Q.

The side-conditions for this rule prevent both the predicates escaping the scope of the module, and repeated definitions of a predicate. The final restriction is not required but reduces the complexity of the modifies clauses for the frame rule.

In fact, the previous function definition rule is a derived rule in our system. It is derived from the standard function definition rule and two new rules for manipulating abstractions:

ABSTRACT WEAKENING

$$\frac{\Lambda; \Gamma \vdash \{P\}C\{Q\}}{\Lambda, \Lambda'; \Gamma \vdash \{P\}C\{Q\}}$$

where $dom(\Lambda')$ and $dom(\Lambda)$ are disjoint

ABSTRACT ELIMINATION

$$\frac{\Lambda, \Lambda'; \Gamma \vdash \{P\}C\{Q\}}{\Lambda; \Gamma \vdash \{P\}C\{Q\}}$$

where the predicate names in P, Q, Γ and Λ are not in $dom(\Lambda')$.

The first, ABSTRACT WEAKENING, allows the introduction of new definitions; and the second, ABSTRACT ELIMINATION allows any unused predicate to be removed.

We derive the abstraction function definition rule by taking the standard function definition rule, and using ABSTRACT WEAKENING on the client code premise and ABSTRACT ELIMINATION on the conclusion to remove the new predicate definitions. We can apply the same technique to the recursive function definition, however we do not require this for our examples.

Next we give one of the standard Hoare logic rules: the rule of consequence. (Of course, we use the other standard rules; space prevents us from listing them here.)

CONSEQUENCE

$$\frac{\Lambda \models P \Rightarrow P' \quad \Lambda; \Gamma \vdash \{P'\}C\{Q'\} \quad \Lambda \models Q' \Rightarrow Q}{\Lambda; \Gamma \vdash \{P\}C\{Q\}}$$

This rule is key to actual use of abstract predicate definitions. We provide the following two axioms concerning abstract predicates:

$$\begin{array}{lll} \text{OPEN} & (\alpha(\overline{x}) \stackrel{\text{def}}{=} P), \Lambda & \models \quad \alpha(\overline{\text{E}}) \Rightarrow P[\overline{\text{E}}/\overline{x}] \\ \text{CLOSE} & (\alpha(\overline{x}) \stackrel{\text{def}}{=} P), \Lambda & \models \quad P[\overline{\text{E}}/\overline{x}] \Rightarrow \alpha(\overline{\text{E}}) \end{array}$$

[3] A positive formula is one where predicate names appear only under an even number of negations. This ensures that a fixed point can be found; this is explained in further detail in §3.4.2

These axioms embody our intuition that if (and only if) an abstract predicate is in scope then we can freely move between its name and its definition.

Next we present the rules for function call and return.

$$\Lambda;\Gamma \vdash \{P[\overline{y}/\overline{x}]\}\ y{=}k(\overline{y})\ \{Q[\overline{y}, y/\overline{x}, ret]\} \quad \text{where } \{P\}k(\overline{x})\{Q\} \in \Gamma$$

$$\Lambda;\Gamma \vdash \{P[x/ret]\}\ \texttt{return}\, x\ \{P\}$$

These rules use the distinguished variable ret to match the return value with its destination variable.

Finally we give the small axioms of separation logic

$$\Lambda;\Gamma \vdash \{\mathrm{E} \mapsto \_\}\ [\mathrm{E}]{=}\mathrm{E}'\ \{\mathrm{E} \mapsto \mathrm{E}'\}$$
$$\Lambda;\Gamma \vdash \{\mathrm{E} \mapsto n \wedge x = m\}\ x{=}[\mathrm{E}]\ \{\mathrm{E}[m/x] \mapsto n \wedge x = n\}$$
$$\Lambda;\Gamma \vdash \{\mathrm{E} \mapsto \_\}\ \texttt{dispose(E)}\ \{\mathit{empty}\}$$
$$\Lambda;\Gamma \vdash \{\mathit{empty} \wedge x = m\}\ x{=}\texttt{cons}(\overline{\mathrm{E}})\ \{x \mapsto \overline{\mathrm{E}}[m/x]\}$$

These refer only to the state that is accessed by the commands. They can typically be extended using the FRAME RULE to refer to a larger state, e.g.

$$\Lambda;\Gamma \vdash \{\mathrm{E} \mapsto \_ * \mathrm{E}_1 \mapsto \mathrm{E}_2\}\ \texttt{dispose(E)}\ \{\mathrm{E}_1 \mapsto \mathrm{E}_2\}\,.$$

3.3 Examples

3.3.1 Connection pool

Our first example is a database connection pool. Constructing a database connection is generally an expensive operation, so this cost is reduced by pooling connections using the object pool design pattern [9]. Programs regularly access several different databases, hence we require multiple connection pools and dynamic instantiation (hence this could not be modelled in the framework of O'Hearn et al. [22]). The connection pool must prevent the connections being used after they are returned: ownership must be transferred between the client and the pool.

We assume a library routine, `consConn`, to construct a database connection. This routine takes a single parameter that specifies the database,[4] and returns a handle to a connection. The specification uses a predicate $conn$ to represent the state of the connection.

$$\{\mathit{empty}\}\ \texttt{consConn(s)}\ \{\mathit{conn}(ret, s)\}$$

We define two abstract predicates for the connection pool module: $cpool$ and $clist$. The $cpool$ predicate is used to represent a connection pool; and the $clist$ predicate is used inside the $cpool$ to represent a list of connection predicates.

$$\mathit{cpool}(x, s) \stackrel{\text{def}}{=} \exists i.x \mapsto i, s * \mathit{clist}(i, s)$$

$$\mathit{clist}(x, s) \stackrel{\text{def}}{=} x \doteq \mathit{null} \vee (\exists ij.x \mapsto i, j\ * \mathit{conn}(i, s) * \mathit{clist}(j, s))$$

where $E \doteq E'$ is a shorthand for $E = E' \wedge \mathit{empty}$.

The connection pool has three operations: construct a pool, `consPool`; get a connection, `getConn`; and free a connection, `freeConn`. These are specified as follows.

$\{\mathit{empty}\}$`consPool(s)`$\{\mathit{cpool}(ret, s)\}$
$\{\mathit{cpool}(x, s)\}$`getConn(x)`$\{\mathit{cpool}(x, s) * \mathit{conn}(ret, s)\}$
$\{\mathit{cpool}(x, s) * \mathit{conn}(y, s)\}$`freeConn(x,y)`$\{\mathit{cpool}(x, s)\}$

We give the implementation of these operations in Figure 2.

We present the proof that the `freeConn` implementation satisifies its specification, which illustrates the use of abstract predicates:

$\{\mathit{cpool}(x, s) * \mathit{conn}(y, s)\}$
$\{\exists i.x \mapsto i, s * \mathit{clist}(i, s) * \mathit{conn}(y, s)\}$
`t=[x];`

[4] In a more realistic implementation, such as JDBC [8], several arguments would be used to specify how to access a database.

```
let
  consPool s =
    (newvar p; p=cons(null,s); return p)
  getConn x =(newvar n,c,l,p; l=[x];
    if (l == null) then
      p=[x+1]; c=consConn(p)
    else (c=[l]; n=[l+1]; dispose(l);
        dispose(l+1); [x]=n);
    return c)
  freeConn x y =
    (newvar t,n; t=[x]; n=cons(y,t); [x]=n)
in
  C
```

Figure 2: Source code for the connection pool

$\{x \mapsto t, s * \mathit{clist}(t, s) * \mathit{conn}(y, s)\}$
`n=cons(y,t);`
$\{x \mapsto t, s * n \mapsto y, t * \mathit{clist}(t, s) * \mathit{conn}(y, s)\}$
`[x]=n`
$\{x \mapsto n, s * n \mapsto y, t * \mathit{clist}(t, s) * \mathit{conn}(y, s)\}$
$\{x \mapsto n, s * \mathit{clist}(n, s)\}$
$\{\mathit{cpool}(x, s)\}$

In this proof the definitions of both $cpool$ and $clist$ are used with OPEN and CLOSE to give the following three implications

$$\mathit{cpool}(x, s) \Rightarrow \exists i.x \mapsto i, s * \mathit{clist}(i, s)$$
$$n \mapsto y, t * \mathit{clist}(t, s) * \mathit{conn}(y, s) \Rightarrow \mathit{clist}(n, s)$$
$$x \mapsto n, s * \mathit{clist}(n, s) \Rightarrow \mathit{cpool}(x, s)$$

These are used with the rule of CONSEQUENCE to complete the proof.

Next we present, and attempt to verify, a fragment of client code using the connection pool. It demonstrates both correct and incorrect usage, which causes the verification to fail. The example calls a function, `useConn`, that uses a connection.

$\{\mathit{cpool}(x, s)\}$
`y = getConn(x);`
$\{\mathit{cpool}(x, s) * \mathit{conn}(y, s)\}$
$\{\mathit{conn}(y, s)\}$
`useConn(y);`
$\{\mathit{conn}(y, s)\}$
$\{\mathit{cpool}(x, s) * \mathit{conn}(y, s)\}$
`freeConn(x,y);`
$\{\mathit{cpool}(x, s)\}$
`useConn(y)`
$\{???\}$

The client gets a connection from the pool, uses it and then returns it. However, after returning it, the client tries to use the connection. This command cannot be validated as the precondition does not contain the $conn$ predicate. Even though this predicate is contained in $cpool$, the client is unable to expand the definition because it is out of scope. This illustrates how abstract predicates capture "ownership transfer". The connection passes from the client into the connection pool stopping the client from accessing it, even though the client has a pointer to the connection.

A connection pool library wants many instances; generally one per database. This can be easily handled by calling `consPool` the required number of times. Assume we have two different databases, `s1` and `s2`.

$\{\mathit{empty}\}$
`y = consPool(s1);`
$\{\mathit{conPool}(y, s1)\}$
`z = consPool(s2);`
$\{\mathit{conPool}(y, s1) * \mathit{conPool}(z, s2)\}$

This code creates two connection pools. The parameter prevents us returning the connection to the incorrect pool.

$\{conPool(y, s1) * conPool(z, s2)\}$
`x = getConn(z);`
$\{conPool(y, s1) * conPool(z, s2) * conn(x, s2)\}$
`freeConn(y,x)`
$\{???\}$

The `freeConn` call can only be validated if `s1 = s2`.[5]

This example has illustrated that abstract predicates capture the notion of "ownership transfer", first presented with the hypothetical frame rule. Abstract predicates additionally deal with dynamic instantiation of a module, which the hypothetical frame rule cannot.

Note: To complete this example we should include a dispose pool function. As it presents no additional interesting difficults we omit it from our exposition.

3.3.2 Malloc and free

The next example is a simple memory manager that allocates variable sized blocks of memory. We use a couple of additional features for handling arrays, described by Reynolds [29]: the iterated separating conjunctions, $\odot_{x=\mathrm{E}_1}^{\mathrm{E}_2}.P$; and a system routine `allocate` that allocates variable sized blocks. Intuitively the iterated separating conjunction, $\odot_{x=\mathrm{E}_1}^{\mathrm{E}_2}.P$, is the expansion

$$P[\mathrm{E}_1/x] * \ldots * P[\mathrm{E}_2/x]$$

where x ranges from E_1 to E_2. If E_2 is less than E_1, it is equivalent to $empty$. More formally its semantics are:

$$\begin{aligned} S,H,I \models_\Delta \odot_{x=\mathrm{E}_1}^{\mathrm{E}_2}.P \stackrel{\mathrm{def}}{=} (&[\![\mathrm{E}_1]\!]_{S,I} = n_1 \wedge [\![\mathrm{E}_2]\!]_{S,I} = n_2) \Rightarrow \\ &((n_1 \leq n_2 \Rightarrow S,H,I \models_\Delta P[n_1/x] * \odot_{x=n_1+1}^{n_2}.P) \\ &\qquad \wedge\ (n_1 > n_2 \Rightarrow S,H,I \models_\Delta empty)) \end{aligned}$$

Returning to the example, consider the following naïve specifications, which demonstrate the difficulties in reasoning about the memory manager:

$\{empty\}$`malloc(n)`$\{\odot_{i=0}^{n-1}.ret + i \mapsto \_\}$
$\{\odot_{i=0}^{n-1}.x + i \mapsto \_\}$`free(x)`$\{empty\}$

The problem is with the specification of `free`: it does not specify how much memory is returned as n is a free variable.

The standard specification [12] of `free` only requires it to deallocate blocks provided by `malloc`. Using abstract predicates we *are* able to provide an adequate specification.

$\{empty\}$`malloc(n)`$\{\odot_{i=0}^{n-1}.ret + i \mapsto \_ * Block(ret, n)\}$
$\{\odot_{i=0}^{n-1}.x + i \mapsto \_ * Block(x, n)\}$`free(x)`$\{empty\}$

The $Block$ predicate is used as a modular certificate that `malloc` actually produced the block. The client can not construct a $Block$ predicate as its definition is not in scope.

Standard implementations of `malloc` and `free` store the block's size in the cell before the allocated block [12]. This can be specified by defining the $Block$ predicate as follows.

$$Block(x, n) \stackrel{\mathrm{def}}{=} x - 1 \mapsto n$$

This allows `free` to determine the quantity of memory returned.[6]

We can give a simple implementations of these routines that call system routines to construct (`allocate`) and dispose (`dispose`) the blocks.[7]

```
malloc n =(newvar x; x=allocate(n+1);
           [x]=n; return x+1)
free x =(newvar n; n=[x-1];
         while(n≥0) (n=n-1; dispose(x+n))
```

Both of their implementations can be verified; here we present the proof of `malloc`:

$\{empty\}$
`x=allocate(n+1);`
$\{\odot_{i=0}^{n}.x + i \mapsto \_\}$
$\{x \mapsto \_ * \odot_{i=1}^{n}.x + i \mapsto \_\}$
`[x]=n;`
$\{x \mapsto n * \odot_{i=1}^{n}.x + i \mapsto \_\}$
`return x+1`
$\{ret - 1 \mapsto n * \odot_{i=1}^{n}.ret - 1 + i \mapsto \_\}$
$\{ret - 1 \mapsto n * \odot_{i=0}^{n-1}.ret + i \mapsto \_\}$
$\{\odot_{i=0}^{n-1}.ret + i \mapsto \_ * Block(ret, n)\}$

The final implication in this proof abstracts the cell containing the block's length, hence the client cannot directly access it. The following code fragment attempts to break this abstraction:

$\{empty\}$
`x=malloc(30);`
$\{\odot_{i=0}^{29}.x + i \mapsto \_ * Block(x, 30)\}$
`[x-1]=15;`
$\{???\}$
`free(x);`

The client attempts to modify the information about the block's size. This would be a clear failure in modularity as the client is dependent on the implementation of $Block$. Fortunately, we are unable to validate the assignment as the pre-condition does not contain $x - 1 \mapsto \_$. Although, the $Block$ contains the cell, the client does not have the definition in scope and hence cannot use it.

O'Hearn, Reynolds and Yang's [22] idealization of a memory manager does not support variable sized blocks. Their specifications can not be extended to cover this without exposing the representation of the block. Additionally, it is impossible for them to enforce that `malloc` must provide the blocks that `free` deallocates without extending the logic.

3.3.3 Permissions reading

O'Hearn [21] has recently given separation logic an ownership, or permissions, interpretation: $\mathrm{E} \mapsto \mathrm{E}'$ is the permission to read, write and dispose the cell at location E. Bornat et al. [5] extend this to allow read sharing. Essentially they annotate the $\mapsto$ relation to express the type of permission it represents: read or total. In the previous example, the $Block$ predicate is the permission to dispose the memory using `free`. Using this permissions reading of separation logic, abstract predicates allow modules to define their own permissions. The concept of ownership transfer can be seen as transferring permission to and from the client.

Consider a ticket machine:

{*empty*}`getTicket()`{*Ticket(ret)*}
{*Ticket (x)*}`useTicket(x)`{*empty*}

To call `useTicket` you must have called `getTicket`; each usage consumes a ticket. Trying to use a ticket twice fails:

{*empty*}
`x = getTicket();`
{*Ticket (x)*}
`useTicket(x);`
{*empty*}
`useTicket(x);`
$\{???\}$

The second call to `useTicket` fails, because the first call removed the $Ticket$.

[5] Given the specification it is always valid to return a connection to a pool if it is to the correct database. A tighter specification could be given to restrict returning to the allocating pool.

[6] More complicated specifications can be used which account for padding and other book keeping.

[7] One could extend the specifications to have an additional memory manager predicate as in the connection pool example.

Any client that is validated against this specification must use the ticket discipline correctly. In fact the module is free to define the ticket in any way, e.g. $Ticket(x) \stackrel{\text{def}}{=} true$. Although this ticket would be logically valid to duplicate, $true * true \Leftrightarrow true$, the client does not know this, and hence cannot.

3.4 Semantics

In the previous section we have informally introduced the notion of abstract predicates and detailed a couple of examples to highlight their use and demonstrate their usefulness. In this section we formalize them precisely and show that the two abstract predicate rules are sound.

3.4.1 Programming language

We assume the usual semantics of separation logic [31] and extend it to handle the functions. A *semantic function environment*, Π, is a finite partial function from function names, k, to a pair of a vector of variable names and a command for the body ($\Pi : k \rightharpoonup (\overline{x}, C)$). An environment is well-formed, Π **ok**, if it only modifies local variables, $\forall \overline{x}, C \in cod(\Pi).\text{modifies}(C) = \emptyset$.

A configuration is defined as a quadruple of a function environment, a command, a stack, and a heap. A terminal configuration is a stack, heap pair or **Fault**. The semantics are given by a recursively defined relation between configurations and terminal configurations presented in Figure 3. We provide additional failure rules for each heap command accessing undefined state:

$$\left.\begin{array}{r}(\Pi, \text{[E]=E}', S, H) \Downarrow \textbf{Fault} \\ (\Pi, x\text{=[E]}, S, H) \Downarrow \textbf{Fault} \\ (\Pi, \texttt{dispose(E)}, S, H) \Downarrow \textbf{Fault}\end{array}\right\} \begin{array}{l}\text{where} \\ [\![\text{E}]\!]_S \notin dom(H)\end{array}$$

and add rules to propagate the **Fault** states in the obvious way.

DEFINITION 3.1 (SAFETY).
$(\Pi, C, S, H) : \textbf{\textit{safe}} \stackrel{\text{def}}{=} \neg((\Pi, C, S, H) \Downarrow \textbf{\textit{Fault}})$
Note: *As we only consider partial correctness, we consider non-termination as safe.*

We have the standard properties required for the soundness of the frame rule [31].

LEMMA 3.2 (SAFETY MONOTONICITY).

$$(\Pi, C, S, H) : \textbf{\textit{safe}} \wedge H' \perp H \Rightarrow (\Pi, C, S, H \circ H') : \textbf{\textit{safe}}$$

LEMMA 3.3 (HEAP LOCALITY).

$$(\Pi, C, S, H_1) : \textbf{\textit{safe}} \wedge (\Pi, C, S, H_1 * H) \Downarrow (S', H') \Rightarrow \exists H_2.H' = H * H_2 \wedge (\Pi, C, S, H_1) \Downarrow (S', H_2)$$

3.4.2 Abstract predicates

Next we define the semantics of an abstract predicate. First we define semantic predicate environments, Δ, as follows:

$$\Delta : \mathcal{A} \rightharpoonup \coprod_{n \in \mathbb{N}} (\mathbb{N}^n \to \mathcal{P}(\mathcal{H}))$$

where $\mathcal{A}$ is the set of predicate names. We restrict our consideration to well-formed environments: each predicate name is mapped to a function of the correct arity, $\Delta(\alpha) : \mathbb{N}^{arity(\alpha)} \to \mathcal{P}(\mathcal{H})$. The reader might have expected the use of $\mathcal{P}(\mathcal{H} \times \mathcal{S} \times \mathcal{I})$, but this breaks substitution as the predicate can depend on variables that are not syntactically free.

The semantics of a predicate is as follows:

$$S, H, I \models_\Delta \alpha(\overline{\text{E}}) \;\Leftrightarrow\; \alpha \in dom(\Delta) \;\wedge\; H \in (\Delta\alpha)[[\![\overline{\text{E}}]\!]_{S,I}]$$

The rest of the semantics are from the standard definition, sketched in §2, with the predicate environment added in the obvious way.

We define the following ordering on semantic predicate environments

$$\Delta \sqsubseteq \Delta' \stackrel{\text{def}}{=} \forall \alpha. \forall \overline{n} : \mathbb{N}^{arity(\alpha)}. \Delta(\alpha) \neq \perp \Rightarrow \Delta(\alpha)(\overline{n}) \subseteq \Delta'(\alpha)(\overline{n})$$

The least upper bound of this order is written $\sqcup$.

LEMMA 3.4. *Well-formed semantic predicate environments form a complete lattice with respect to* $\sqsubseteq$.

LEMMA 3.5. *Formulae only depend on the predicate names they mention, i.e. if* Δ *defines all the predicate names in P, and* Δ *and* Δ' *are disjoint, then*

$$\forall S, H, I. \quad S, H, I \models_\Delta P \Leftrightarrow S, H, I \models_{\Delta \sqcup \Delta'} P$$

LEMMA 3.6. *Positive formulae are monotonic with respect to semantic predicate environments, i.e. if* P *is a positive formula,*

$$\Delta \sqsubseteq \Delta' \;\wedge\; S, H, I \models_\Delta P \;\Rightarrow\; S, H, I \models_{\Delta'} P$$

Now let us consider the construction of a semantic predicate environment from an abstract one, Λ. The abstract predicate environment does not, necessarily, define every predicate, so constructing a solution requires additional semantic definitions, Δ, to fill the holes. We use the following function to generate a fixed point:

$$step_{(\Delta,\Lambda)}(\Delta_n) \stackrel{\text{def}}{=} \lambda\alpha \in dom(\Lambda).\lambda\overline{n} \in \mathbb{N}^{arity(\alpha)}. \{H | S, H, I \models_{\Delta_n \sqcup \Delta} \Lambda(\alpha)[\overline{n}]\}$$

where Λ are the definitions we want to solve; Δ are the predicates not defined in Λ; and Δ_n is an approximation to the solution. $step$ is monotonic on predicate environments, because of Lemma 3.6 and that all the definitions are positive. Hence by Tarski's theorem and Lemma 3.4 we know a least fixed point always exists. We write $[\![\Lambda]\!]_\Delta$ for the least fixed point of $step_{\Delta,\Lambda}$.
Note: $step$ is not Scott-continuous. This does not cause any problems because we only need consider the properties of the least fixed point, rather than its construction.

Consider the set of all solutions of Λ of the form $\Delta \sqcup [\![\Lambda]\!]_\Delta$:

$$\text{close}(\Lambda) \stackrel{\text{def}}{=} \{\Delta \sqcup [\![\Lambda]\!]_\Delta \mid dom(\Delta) = \mathcal{A} \setminus dom(\Lambda)\}$$

This function has two properties.

LEMMA 3.7. *Adding new predicate definitions refines the set of possible semantic predicate environments, i.e.*

$$close(\Lambda) \supseteq close(\Lambda, \Lambda')$$

LEMMA 3.8. *The removal of predicate definitions does not affect predicates that do not use them. Given* Λ *which is disjoint from* Λ' *and does not mention predicate names in its domain; we have*

$$\forall \Delta \in close(\Lambda). \exists \Delta' \in close(\Lambda, \Lambda'). \; \Delta \upharpoonright dom(\Lambda') = \Delta' \upharpoonright dom(\Lambda')$$

where $f \upharpoonright S$ *is* $\{a \mapsto b | a \mapsto b \in f \wedge a \notin dom(S)\}$

We define validity wrt an abstract predicate environment, written $\Lambda \models P$, as follows:

$$\forall S, H, I, \Delta \in \text{close}(\Lambda). \; S, H, I \models_\Delta P$$

THEOREM 3.9. OPEN *and* CLOSE, *i.e.*

$$\alpha(\overline{x}) \stackrel{\text{def}}{=} P, \Lambda \models \alpha(\overline{E}) \Rightarrow P[\overline{E}/\overline{x}]$$
$$\alpha(\overline{x}) \stackrel{\text{def}}{=} P, \Lambda \models P[\overline{E}/\overline{x}] \Rightarrow \alpha(\overline{E}),$$

are valid.

Function definition

$$\frac{(\Pi[k_1 \mapsto (\overline{x_1}, C_1), \ldots, k_n \mapsto (\overline{x_n}, C_n)], C, S, H) \Downarrow (S', H')}{(\Pi, \texttt{let}\ k_1\ \overline{x_1}\texttt{=}C_1, \ldots, k_n\ \overline{x_n}\texttt{=}C_n\ \texttt{in}\ C), S, H) \Downarrow (S', H')}$$

New variable

$$\frac{(\Pi, C, S[x \mapsto nil], H) \Downarrow (S_1, H_1)}{(\Pi, \texttt{newvar}\ x; C, S, H) \Downarrow (S_1[x \mapsto S(x)], H_1)}$$

While1

$$\frac{[\![B]\!]_S = false}{(\Pi, \texttt{while}\ B\ C, S_1, H_1) \Downarrow (S_1, H_1)}$$

While2

$$\frac{[\![B]\!]_S = true \quad (\Pi, C; \texttt{while}\ B\ C, S_1, H_1) \Downarrow (S_2, H_2)}{(\Pi, \texttt{while}\ B\ C, S_1, H_1) \Downarrow (S_2, H_2)}$$

Function call

$$\frac{(\Pi, C, \overline{x} \mapsto [\![\overline{y}]\!]_S, H) \Downarrow (S', H') \quad \Pi(k) = \overline{x}, C}{(\Pi, x\texttt{=}k(\overline{y}), S, H) \Downarrow (S[x \mapsto [\![ret]\!]_{S'}], H')}$$

Sequence

$$\frac{(\Pi, C_1, S_1, H_1) \Downarrow (S_2, H_2) \quad (\Pi, C_2, S_2, H_2) \Downarrow (S_3, H_3)}{(\Pi, C_1; C_2, S_1, H_1) \Downarrow (S_3, H_3)}$$

If1

$$\frac{[\![B]\!]_S = true \quad (\Pi, C_1, S, H) \Downarrow (S_1, H_1)}{(\Pi, \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, S, H) \Downarrow (S_1, H_1)}$$

If2

$$\frac{[\![B]\!]_S = false \quad (\Pi, C_2, S, H) \Downarrow (S_1, H_1)}{(\Pi, \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, S, H) \Downarrow (S_1, H_1)}$$

Read	$(\Pi, x\texttt{=}[\mathrm{E}], S, H) \Downarrow (S[x \mapsto n], H)$	where $H([\![\mathrm{E}]\!]_S) = n$
Write	$(\Pi, [\mathrm{E}]\texttt{=}\mathrm{E}', S, H) \Downarrow (S, H[n \mapsto n'])$	where $[\![\mathrm{E}]\!]_S = n$, $n \in dom(H)$ and $[\![\mathrm{E}']\!]_S = n'$
Cons	$(\Pi, x\texttt{=}\texttt{cons}(\overline{\mathrm{E}}), S, H) \Downarrow (S[x \mapsto n], H[n \mapsto \overline{n}])$	where $\lvert\overline{\mathrm{E}}\rvert = n'$, $\{n, \ldots, n+n'\} \perp dom(H)$ and $[\![\overline{\mathrm{E}}]\!]_S = \overline{n}$
Dispose	$(\Pi, \texttt{dispose}(\mathrm{E}), S, H) \Downarrow (S, H')$	where $[\![\mathrm{E}]\!]_S = n$, $H'[n \mapsto n'] = H$ and $n \notin dom(H')$
Assign	$(\Pi, x\texttt{=}\mathrm{E}, S, H) \Downarrow (S[x \mapsto n], H)$	where $[\![\mathrm{E}]\!]_S = n$
Return	$(\Pi, \texttt{return}\,\mathrm{E};, S, H) \Downarrow (S[ret \mapsto [\![\mathrm{E}]\!]_S], H)$	

Figure 3: Operational semantics

3.4.3 Judgements

We are now in a position to define a semantics for our reasoning system. We write $\Lambda; \Gamma \models \{P\}C\{Q\}$ to mean that, if every specification in Γ is true of a function environment, and every abstract predicate definition in Λ is true of a predicate environment, then so is $\{P\}C\{Q\}$:

$$\Lambda; \Gamma \models \{P\}C\{Q\} \stackrel{\text{def}}{=} \forall \Delta \in \text{close}(\Lambda), \Pi.\ \Pi\ \mathbf{ok} \wedge (\Delta \models_\Pi \Gamma) \Rightarrow \Delta \models_\Pi \{P\}C\{Q\}$$

where

$$\Delta \models_\Pi \Gamma \stackrel{\text{def}}{=} \forall \{P\}k\{Q\} \in \Gamma.\Pi(k) = (\overline{x}, C) \Rightarrow \Delta \models_\Pi \{P\}C\{Q\}$$

$$\Delta \models_\Pi \{P\}C\{Q\} \stackrel{\text{def}}{=} \forall S, H, I.\ S, H, I \models_\Delta P \Rightarrow ((\Pi, C, S, H) : \mathbf{safe} \wedge ((\Pi, C, S, H) \Downarrow (S', H') \Rightarrow S', H', I \models_\Delta Q))$$

Given this definition we can show that the two new rules for abstract predicates are sound.

THEOREM 3.10. *Abstract weakening is sound.*

PROOF. Direct consequence of definition of judgements and Lemma 3.7. □

THEOREM 3.11. *Abstract elimination is sound.*

PROOF. Follows from Lemmas 3.5 and 3.8. □

4. A JAVA-LIKE LANGUAGE

In the previous section we have shown how abstract predicates can be used with separation logic to provide a powerful but intuitive framework to reason about a language with first-order functions or procedures. We now turn our attention to another form of modularity: class-based objects.

More precisely we shall consider the problems of reasoning about a fragment of Java. We will consider a simple subset of Java based on Middleweight Java (MJ) [3]. We restrict MJ's expressions to be

Program
prog ::= *cldef*$_1$ ... *cldef*$_n$; $\overline{s}$
Class definition
cldef ::= `class` C `extends` D {$\overline{\textit{fdef}}$ $\overline{\textit{mdef}}$}
Method definition
mdef ::= C m(C_1 x_1, ..., C_n x_n) { $\overline{s}$ `return` x`;` }
Field definition
fdef ::= C f`;`
Expressions
E ::= x | `null`
Statements
s ::= x = y.f`;` | x = (C)y`;` | x = `new` C`();`
| x.f = E`;` | x = y.m($\overline{\mathrm{E}}$)`;` | C x`;`
| {$\overline{s}$} | `;` | `if` (E == E) {$\overline{s}$} `else` {$\overline{s}$}

Figure 4: Syntax of MJ subset

stack variables and `null`,[8] and remove constructors. We present the full syntax in Figure 4. We write f and m to range over field and method names respectively. We use C, D to range over class names, and x, y to range over variable names.

Consider giving a separation logic to this language: clearly we require the "points to" relation to describe fields.[9] The new assertion "field points to", written $x.f \mapsto y$, means the field f of the object x contains the value y. We also use the predicate $x : C$ to mean that x points to an object of class C: it is actually a C not just a subtype of C.

Now consider a motivational example [1] of a `Cell` class and a subclass that has backup, `Recell`, presented in Figure 5. The specifications of the `set` methods are:

$\{\texttt{this}.cnts \mapsto \_\}$	$\{\texttt{this}.cnts \mapsto X * \texttt{this}.bak \mapsto \_\}$
`Cell.set(n)`	`Recell.set(n)`
$\{\texttt{this}.cnts \mapsto n\}$	$\{\texttt{this}.cnts \mapsto n * \texttt{this}.bak \mapsto X\}$

These specifications have two problems: (a) they have no encapsulation; and (b) they do not respect behavioural subtyping.

(a) From the specification the client knows which field is used to

[8] This restriction is required as separation logic requires expressions to be pure: they cannot access the heap, i.e. $x.f_1.f_2$ is not allowed.

[9] An alternative approach would be to use the heap primitive as a whole object [18]. However, being able to split an object allows for more flexible reasoning.

```
class Cell {
  Object cnts;
  void set(Object n) {this.cnts = n;}
  Object get() {Object t;
    t = this.cnts; return t;}}
class Recell extends Cell {
  Object bak;
  void set(Object n) {
    Object t; t = this.cnts;
    this.bak = t; this.cnts = n;}}
```

Figure 5: Source code for Cell and Recell classes

store the contents. Clearly we need a greater level of abstraction. Using an abstract predicate allows us to encapsulate the object's state. We can write the `Cell`'s `set` specification as:

$$\{Val_{Cell}(\texttt{this}, \_)\}\ \texttt{Cell.set(n)}\ \{Val_{Cell}(\texttt{this}, n)\}$$

and define the abstract predicate as

$$Val_{Cell}(\texttt{this}, x) \stackrel{\text{def}}{=} \texttt{this}.cnts \mapsto x$$

and scope it to the `Cell` class. This stops the `Cell`'s client using the field directly as it is hidden in the abstract predicate.

(b) Using standard behavioural subtyping [17], to allow dynamic dispatch, the `Recell`'s specification needs to be compatible with the `Cell`'s, i.e. we require the following two implications to hold

$$pre(\texttt{Cell}, \texttt{set}) \Rightarrow pre(\texttt{Recell}, \texttt{set}) \quad (1)$$

$$post(\texttt{Recell}, \texttt{set}) \Rightarrow post(\texttt{Cell}, \texttt{set}) \quad (2)$$

where $pre(C, m)$ denotes the pre-condition for the method m in class C, and $post$ denotes the post-condition.

Given the earlier specifications, these implications can never hold as they require a one element heap to be the same size as a two element heap. What about abstract predicates? The specification for `Recell` must use a different abstract predicate to `Cell` as it has a different body, i.e.

$$\{Val_{Recell}(\texttt{this}, X, \_)\}\texttt{Recell.set(n)}\{Val_{Recell}(\texttt{this}, n, X)\}$$

with the obvious definition for Val_{Recell}. Unfortunately the predicates are treated parametrically; no implications hold between them.

As it stands, abstract predicates do not, by themselves, help with behavioural subtyping. They provide support for encapsulation but not inheritance. In an object-oriented setting we require predicates to have multiple definitions, hence we introduce *abstract predicate families* where the families are sets of definitions indexed by class. Abstract predicate family instances[10] are written $\alpha(x; \vec{v})$ to indicate that the object x satisfies *one* definition from the abstract predicate family α with arguments $\vec{v}$. The particular definition satisfied depends on the dynamic type of x. In object-oriented programming an object could be from one of many classes; abstract predicate families provide a similar choice of predicate definitions when considering their behaviour.

We define abstract predicate family definitions, Λ_f, with the following syntax.

$$\Lambda_f := \epsilon \mid \alpha_C \stackrel{\text{def}}{=} \lambda(x; \overline{x})P, \Lambda_f$$

Λ_f is well-formed if it has at most one entry for each predicate and class name pair, and the free variables of the body, P, are in its argument list, $x; \overline{x}$. We treat Λ_f as a function from predicate and class name pairs to formulae. Each entry corresponds to the definition of an abstract predicate family for a particular class.

Our example shows the need to alter the arity of the predicate to reflect casting; the `Recell`'s predicate has three arguments while the `Cell`'s only has two. Hence we provide the following pair of implications:

$$\text{WIDEN} \qquad \Lambda_f \models \alpha(x; \overline{x}) \Rightarrow \exists \overline{y}.\alpha(x; \overline{x}, \overline{y})$$

$$\text{NARROW} \qquad \Lambda_f \models \exists \overline{y}.\alpha(x; \overline{x}, \overline{y}) \Rightarrow \alpha(x; \overline{x})$$

If we give a predicate more variables than its definition requires, it ignores them, and if too few, it treats the missing arguments as existentially quantified. This leads to our definition of substitution onto a predicate definition,

$$(\lambda(x; \overline{x}).P)[\mathrm{E}; \overline{\mathrm{E}}] \stackrel{\text{def}}{=} \begin{cases} P[\mathrm{E}/x, \overline{\mathrm{E}_1}/\overline{x}] & |\overline{\mathrm{E}_1}| = |\overline{x}| \text{ and } \overline{\mathrm{E}} = \overline{\mathrm{E}_1}, \overline{\mathrm{E}_2} \\ \exists \overline{y}.P[\mathrm{E}/x, (\overline{\mathrm{E}}, \overline{y})/\overline{x}] & |\overline{\mathrm{E}}, \overline{y}| = |\overline{x}| \end{cases}$$

This definition of substitution can then be used to give the families' version of OPEN and CLOSE.

$$\text{OPEN} \qquad \Lambda_f \models (x : C \wedge \alpha(x; \overline{x})) \Rightarrow \Lambda_f(\alpha, C)[x; \overline{x}]$$

$$\text{CLOSE} \qquad \Lambda_f \models (x : C \wedge \Lambda_f(\alpha, C)[x; \overline{x}]) \Rightarrow \alpha(x; \overline{x})$$

where $\alpha, C \in dom(\Lambda_f)$.

To OPEN or CLOSE a predicate we must know which class contains the definition, and must have that version of the predicate in scope.

Note: We can OPEN predicates at incorrect arities as the substitution will correctly manipulate the arguments. An alternative approach would be to restrict opening to the correct arity, and use WIDEN and NARROW to get the correct arity. However, this approach complicates the semantics.

4.1 Proof rules

In this section we define a set of Hoare-style proof rules for reasoning about MJ programs. The judgements take the following form:

$$\Lambda_f; \Gamma \vdash \{P\}\overline{s}\{Q\}$$

where Γ is a set of assertions about methods. They have the following form:

$$\Gamma \quad := \quad \epsilon \mid \{P\}C.m(\overline{x})\{Q\}, \Gamma$$

A well-formed method environment, $\vdash \Gamma\ \mathbf{wf}$, defines each method and class name pair only once and has the following three properties:

1. The pre- and post-conditions can only contain free program variables in the argument list, `this` and ret; i.e.

$$\forall \{P\}C.m(\overline{x})\{Q\} \in \Gamma.\text{FPV}(P) \subseteq (\{\texttt{this}\} \cup \overline{x}) \wedge \text{FPV}(Q) \subseteq (\{\texttt{this}, ret\} \cup \overline{x})$$

2. A method can only modify local variables; there are no global variables and arguments cannot be modified, i.e.

$$\forall \{P\}C.m(\overline{x})\{Q\} \in \Gamma.\text{modifies}(mbody(C, m)) = \emptyset$$

   where $mbody(C, m)$ returns the body of method m in class C.

3. Subtypes must have compatible specifications with their supertypes, i.e.

$$\forall \{P_C\}C.m(\overline{x})\{Q_C\} \in \Gamma.\ D \prec C \Rightarrow \{P_D\}D.m\{Q_D\} \in \Gamma \wedge (\vdash \{P_C\}\_\{Q_C\} \Rightarrow \{P_D\}\_\{Q_D\})$$

[10] In the module system the concept of abstract predicate instance, and the abstract predicate are conflated, but here as we have multiple definitions the distinction must be kept.

DEFINITION 4.1 (SPECIFICATION COMPATIBILITY). *We define specification compatiblity,* $\vdash \{P_C\}\_\{Q_C\} \Rightarrow \{P_D\}\_\{Q_D\}$, *as* $\forall \overline{s}$. *if* $\Lambda; \Gamma \vdash \{P_D\}\overline{s}\{Q_D\}$ *is derivable from* $\Lambda; \Gamma \vdash \{P_C\}\overline{s}\{Q_C\}$ *using only the structural rules:* CONSEQUENCE, AUXILIARY VARIABLE ELIMINATION *and* VARIABLE SUBSTITUTION.[11]

This is more general than the behavioural subtyping rules as it allows manipulation of auxiliary variables. In fact, if the derivation only uses the rule of CONSEQUENCE, specification compatibilty degenerates to behavioural subtyping.

Now let us consider the method call rule:

METHOD CALL

$$\Lambda; \Gamma \vdash \left\{ \begin{array}{l} P[x, \overline{y}/\texttt{this}, \overline{x}] \\ \quad \wedge\ x\,!=\texttt{null} \end{array} \right\} y{=}x.m(\overline{y}) \{Q[x, y, \overline{y}/\texttt{this}, ret, \overline{x}]\}$$

where x has static type C and $\{P\}C.m(\overline{x})\{Q\} \in \Gamma$

The method call rule only needs to consider the static type of the receiver, because we have restricted ourselves to methods that are specification compatible.[12]

The rules for checking the whole program deserve some attention.

CLASS

$$\frac{\begin{array}{c} \Lambda_f; \Gamma \vdash \{P_n \wedge \texttt{this} : C\} mbody(C, m_n) \{Q_n\} \\ \vdots \\ \Lambda_f; \Gamma \vdash \{P_1 \wedge \texttt{this} : C\} mbody(C, m_1) \{Q_1\} \end{array}}{\Lambda_f; \Gamma \vdash \{P_1\}C.m_1(\overline{x_1})\{Q_1\}, \ldots, \{P_n\}C.m_n(\overline{x_n})\{Q_n\}}$$

PROGRAM

$$\frac{\Lambda_{f_1}; \Gamma \vdash \Gamma_1 \quad \ldots \quad \Lambda_{f_n}; \Gamma \vdash \Gamma_n \quad \emptyset, \Gamma \vdash \{P\}\overline{s}\{Q\}}{\vdash \{P\}\mathit{cldef}_1 \ldots \mathit{cldef}_n;\ \overline{s}\{Q\}}$$

where Γ_1 is the method specifications of the methods defined in and inherited into cldef_1; …; Γ_n is induced by cldef_n; $\Gamma = \Gamma_1, \ldots, \Gamma_n$; and $\Lambda_{f_1}, \ldots, \Lambda_{f_n}$ have disjoint domains.

These two rules correspond to the abstract function definition from the previous section. They enforce that each method is checked with the predicate definitions associated to its class.[13] Inherited methods must be rechecked with the new predicate definitions for the class that inherits them. This is because when we check the method bodies in the *class* rule, we add to the pre-condition `this` : C. Without this we would not be able to open or close the abstract predicate families.

Again we have the two rules for introducing and eliminating abstract predicate families.

ABSTRACT WEAKENING

$$\frac{\Lambda_f; \Gamma \vdash \{P\}C\{Q\}}{\Lambda_f, \Lambda'_f; \Gamma \vdash \{P\}C\{Q\}}$$

where $dom(\Lambda'_f)$ and $dom(\Lambda)$ are disjoint.

ABSTRACT ELIMINATION

$$\frac{\Lambda_f, \Lambda'_f; \Gamma \vdash \{P\}C\{Q\}}{\Lambda_f; \Gamma \vdash \{P\}C\{Q\}}$$

where the predicates names in P, Q, Γ and Λ are not in $dom_a(\Lambda'_f)$ and define $dom_a(\Lambda_f)$ as $\{\alpha | (\alpha, C) \in dom(\Lambda_f)\}$.

[11] The frame rule could be included in this list if $\overline{s}$ is restricted to terms that modify no variables.

[12] We could present additional rules that do not rely on the subtyping constraint, but they would only serve to complicate the presentation and wouldn't illustrate anything interesting.

[13] We assume these definitions will be provided during the proof, and provide no explicit syntax for them.

Abstract predicate families are less symmetric than abstract predicates: weakening allows the introduction for a particular class and predicate, while elimination requires the entire family of definitions to be removed, i.e. must remove all the classes' definitions for a predicate. This is because it is not possible to give a simple syntactic check for which parts, i.e. classes, of a family will be used.

Finally we give the rules for field access, field write, and object construction, which are similar to their equivalents in the module system:

$$\Lambda_f; \Gamma \vdash \{x.f \mapsto \_\}\ x.f{=}\mathrm{E}'\ \{x.f \mapsto \mathrm{E}'\}$$

$$\Lambda_f; \Gamma \vdash \left\{ \begin{array}{l} y.f \mapsto n \\ \wedge\ \ x = m \end{array} \right\} x{=}y.f \left\{ \begin{array}{l} y[m/x].f \mapsto n \\ \qquad \wedge\ \ x = n \end{array} \right\}$$

$$\Lambda_f; \Gamma \vdash \{\mathit{empty}\}\ x{=}\texttt{new}\ \ C()\ \{x.f_1 \mapsto \_ * \ldots * x.f_n \mapsto \_ \wedge x : C\}$$
where C has fields $f_1 \ldots f_n$

4.2 Example: Cell/Recell

Let us return to our original motivating example. We define an abstract predicate family, *Val*, with the definitions for `Cell` and `Recell` given earlier.

We have to validate four methods: `Cell.set`, `Cell.get`, `Recell.set` and `Recell.get`. Even though the bodies of `Cell.get` and `Recell.get` are the same, we must validate both, because they have different predicate definitions.

We give the proof for `Recell.set`.

$\{\mathit{Val}(\texttt{this}; X, \_) \wedge \texttt{this} : \mathit{Recell}\}$
$\{\texttt{this}.\mathit{cnts} \mapsto X * \texttt{this}.\mathit{bak} \mapsto \_ \wedge \texttt{this} : \mathit{Recell}\}$
`t = this.cnts;`
$\{\texttt{this}.\mathit{cnts} \mapsto X * \texttt{this}.\mathit{bak} \mapsto \_ \wedge \texttt{this} : \mathit{Recell} \wedge X = t\}$
`this.bak = t;`
$\{\texttt{this}.\mathit{cnts} \mapsto X * \texttt{this}.\mathit{bak} \mapsto t \wedge \texttt{this} : \mathit{Recell} \wedge X = t\}$
`this.cnts = n;`
$\{\texttt{this}.\mathit{cnts} \mapsto n * \texttt{this}.\mathit{bak} \mapsto t \wedge \texttt{this} : \mathit{Recell} \wedge X = t\}$
$\{\texttt{this}.\mathit{cnts} \mapsto n * \texttt{this}.\mathit{bak} \mapsto X \wedge \texttt{this} : \mathit{Recell}\}$
$\{\mathit{Val}(\texttt{this}; n, X)\}$

The other method bodies are all easily verifiable.

Additionally, we must prove the method specifications are compatible. The compatiblity of the `set` method follows from the rule of CONSEQUENCE and AUXILIARY VARIABLE ELIMINATION.

$$\frac{\vdash \{\mathit{Val}(\texttt{this}; X, \_)\}\_\{\mathit{Val}(\texttt{this}; n, X)\}}{\vdash \{\mathit{Val}(\texttt{this}; \_, \_)\}\_\{\mathit{Val}(\texttt{this}; n, \_)\}} \\ \vdash \{\mathit{Val}(\texttt{this}; \_)\}\_\{\mathit{Val}(\texttt{this}; n)\}$$

The `get` methods have the same specification, so are obviously compatible.

A client that uses this code does not need to worry about dynamic dispatch, because of the behavioural subtyping constraints. Consider the following method:

```
m(Cell c) {
  c.set(c);
}
```

This code simply sets the Cell to point to itself. The code is specified as

$\{\mathit{Val}(c; \_)\}$ `m(Cell c)` $\ldots$ $\{\mathit{Val}(c; c)\}$

Now consider calling `m` with a `Recell` argument.

$\{\mathit{empty}\}$
`Recell r = new Recell(x);`
$\{\mathit{Val}(r; x, \_)\}$
$\{\mathit{Val}(r; \_)\}$
`m(r);`
$\{\mathit{Val}(r; r)\}$
$\{\mathit{Val}(r; r, \_)\}$

We use CONSEQUENCE to cast the Val predicate to have the correct arity. We need not consider dynamic dispatch at all because of behavioural subtyping.

Note: The specification of method `m` is weaker than we might like. Based on the implementation we might expect the post-condition $\{Val(r; r, x)\}$. However, there are several bodies that satisfy `m`'s specification: for example `c.set(x);c.set(c);`. We can set the `Cell` to have any value, as long as the last value we set is the `Cell` itself. This body acts identically on a `Cell` to the previous body, however on a `Recell` acts differently. Hence only using the specification we can not infer the tighter post-condition. This could be deduced if `m` was specified for a `Recell` as well.

4.3 Semantics

In this section we consider the extensions to the semantics of §3.4 sufficient to model abstract predicate families. MJ has been defined formally elsewhere [3]. First we shall make some small changes to the basics of the separation logic setting:

DEFINITION 4.2. *A heap, H, is composed of two functions, $H = (H_v, H_t)$: the first, H_v, maps pairs of object identifiers and field names, (oid, f), to values, val; and the second, H_t, maps object identifiers to class names, C. We use $H(oid, f)$ to refer to the value given by the first function, $H_v(oid, f)$, and $H(oid)$ to refer to the value given by the second function, $H_t(oid)$.*

(We make the obvious alterations to the semantics to deal with the new heap definition.) This definition allows a heap to contain only some fields of an object. This new definition also separates the type information from the value information in the heap.

We use the following two definitions to give the partial commutative heap composition monoid

$$(H'_v, H'_t) * (H''_v, H''_t) \stackrel{\text{def}}{=} (H'_v \circ H''_v, H'_t)$$

and is defined iff $dom(H'_v) \bot dom(H''_v)$ and $H'_t = H''_t$ where $\circ$ is composition of disjoint partial functions.

The semantic predicate environment as defined in §3.4 has to be extended to handle the arity changes that predicate families require. We define a semantic predicate family environment as

$$\Delta_f : \mathcal{A} \times \mathbb{C} \rightharpoonup \left(\mathbb{N}^+ \to \mathbb{P}(\mathcal{H})\right)$$

This is a partial function from pairs of predicate and class name to semantic definition. An abstract predicate family is defined for all arities, so the semantic definition must be a function from *all* tuples of non-zero arity. This semantically supports the change in arity required by WIDEN and NARROW.

We can now give the semantics of the new assertions as follows

$$S, H, I \models_{\Delta_f} \mathrm{E}.f \mapsto \mathrm{E}' \Leftrightarrow H([\![\mathrm{E}]\!]_{S,I}, f) = [\![\mathrm{E}']\!]_{S,I} \wedge dom(H) = \{[\![\mathrm{E}]\!]_{S,I}, f\}$$

$$S, H, I \models_{\Delta_f} \mathrm{E} : C \;\Leftrightarrow\; H([\![E]\!]_{S,I}) = C$$

$$S, H, I \models_{\Delta_f} \alpha(\mathrm{E}; \overline{\mathrm{E}}) \Leftrightarrow H \in (\Delta_f(\alpha, C))[[\![\mathrm{E}; \overline{\mathrm{E}}]\!]_{S,I}] \wedge H([\![\mathrm{E}]\!]_{S,I}) = C$$

The field "points to" relation, $\mathrm{E}.f \mapsto \mathrm{E}'$, holds if the heap consists of a single field, f, of the object $[\![\mathrm{E}]\!]_{S,I}$ and has the value $[\![\mathrm{E}']\!]_{S,I}$. $\mathrm{E} : C$ is true if the heap types $[\![\mathrm{E}]\!]_{S,I}$ as class C. $\alpha(\mathrm{E}; \overline{\mathrm{E}})$ holds for some heap H, where $[\![\mathrm{E}]\!]_{S,I}$ has class C, iff H satisfies the predicate definition for C, given arguments $[\![\mathrm{E}, \overline{\mathrm{E}}]\!]_{S,I}$, in the predicate family α.

To ensure that WIDEN and NARROW hold we restrict our attention to *argument refineable* environments.

DEFINITION 4.3 (ARGUMENT REFINEABLE). *A semantic predicate family environment is said to be argument refineable if adding an argument can not increase, or "decrease", the set of accepting states, i.e.*

$$AR(\Delta_f) \Leftrightarrow \forall \alpha, n, \overline{n}.\ \Delta_f(\alpha)[n; \overline{n}] = \bigcup_{n'} \Delta_f(\alpha)[n; \overline{n}, n']$$

PROPOSITION 4.4. *Argument refinement coincides precisely with* WIDEN *and* NARROW*: $\forall S, H, I, \alpha, n, \overline{n}$.*

$$AR(\Delta_f) \Longleftrightarrow (S, H, I \models_{\Delta_f} \alpha(n; \overline{n}) \Leftrightarrow \exists n'.\alpha(n; \overline{n}, n'))$$

We can define an order on semantic predicate families environments, i.e

$$\Delta_f \sqsubseteq \Delta'_f \stackrel{\text{def}}{=} \forall \alpha, C, n, \overline{n}.\Delta_f(\alpha, C)[n; \overline{n}] \subseteq \Delta_f(\alpha, C)[n; \overline{n}]$$

Again, the least upper bound of the order is written $\sqcup$. Lemmas 3.4 and 3.6 can be extended to semantic predicate family environments as follows:

LEMMA 4.5. *Argument refineable predicate family environments form a complete lattice with respect to $\sqsubseteq$.*

LEMMA 4.6. *Positive formulae are monotonic with respect to semantic predicate family environments*

$$\Delta_f \sqsubseteq \Delta'_f \;\wedge\; S, H, I \models_{\Delta_f} P \;\Rightarrow\; S, H, I \models_{\Delta'_f} P$$

However extending Lemma 3.5 is less straight forward, as it is not possible to tell which predicate name, class name pairs are used in a formula.

LEMMA 4.7. *Formulae only depend on the abstractions they mention. If Δ_f contains all the abstractions in P, and $dom_a(\Delta_f) \cap dom_a(\Delta'_f) = \emptyset$, then*

$$\forall S, H, I.\ S, H, I \models_{\Delta_f} P \Leftrightarrow S, H, I \models_{\Delta_f \sqcup \Delta'_f} P$$

Now let us consider the construction of semantic predicate family environments from their abstract syntactic counterparts. We define a new function, $stepf$, that accounts for the first argument's type and uses the special substitution,

$$stepf_{(\Lambda_f, \Delta_f)}(\Delta'_f)(\alpha, C)[n; \overline{n}] \stackrel{\text{def}}{=} \{H | H(n) = C \wedge S, H, I \models_{\Delta_f \cup \Delta'_f} \Lambda_f(\alpha, C)[n; \overline{n}]\}$$

This function is monotonic on predicate family environments, because of Lemma 4.6 and that all the predicate definitions are positive. Hence by Lemma 4.5 and Tarski's theorem we know a fixed point must always exist. We write $[\![\Lambda_f]\!]_{\Delta_f}$ as the least fixed point of $stepf_{(\Lambda_f, \Delta_f)}$.

LEMMA 4.8. *$stepf$ produces argument refineable results.*

Consider the following set of solutions:

$$\{[\![\Lambda_f]\!]_{\Delta_f} \sqcup \Delta_f \mid (\mathcal{A} \times \mathbb{C}) \setminus dom(\Delta_f) = dom(\Lambda_f) \wedge AR(\Delta_f)\}$$

This satisfies the analogues of Lemmas 3.7 and 3.8.

LEMMA 4.9. *Adding new predicate definitions refines the set of possible semantic predicate environments.*

$$close(\Lambda_f) \supseteq close(\Lambda_f, \Lambda'_f)$$

LEMMA 4.10. *The removal of predicate definitions does not affect predicates that do not use them, i.e. given Λ_f which is disjoint from Λ'_f and does not mention predicates in its domain; we have*

$$\forall \Delta \in close(\Lambda_f).\exists \Delta'_f \in close(\Lambda_f, \Lambda'_f).\ \Delta_f \upharpoonright dom(\Lambda'_f) = \Delta'_f \upharpoonright dom(\Lambda'_f)$$

where $f \upharpoonright S$ is $\{a \mapsto b | a \mapsto b \in f \wedge a \notin dom(S)\}$

Validity is defined identically to the previous section, i.e.

$$\Lambda_f \models P \stackrel{\text{def}}{=} \forall S, H, I, \Delta_f \in \text{close}(\Lambda_f).S, H, I \models_{\Delta_f} P$$

THEOREM 4.11. OPEN *and* CLOSE*, i.e.*

$$\Lambda_f \models \alpha(E; \overline{E}) \wedge E : C \Rightarrow \Lambda_f(\alpha, C)[E, \overline{E}/x, \overline{x}]$$
$$\Lambda_f \models \Lambda_f(\alpha, C)[E, \overline{E}/x, \overline{x}] \wedge E : C \Rightarrow \alpha(E; \overline{E})$$

where $(\alpha, C) \in dom(\Lambda_f)$*, are valid.*

THEOREM 4.12. WIDEN *and* NARROW*, i.e.*

$$\Lambda_f \models \alpha(E; \overline{E}) \Rightarrow \exists X.\alpha(E; \overline{E}, X)$$
$$\Lambda_f \models \alpha(E; \overline{E}, E') \Rightarrow \alpha(E; \overline{E}),$$

are valid.

4.3.1 Judgements

We are now in a position to define the semantics for our reasoning system. We write $\Lambda_f; \Gamma \models \{P\}C\{Q\}$ to mean if every specification in Γ is true of a method environment, and every abstract predicate family in Λ_f is true of a predicate family environment, then so is $\{P\}C\{Q\}$, i.e.

$$\Lambda_f; \Gamma \models \{P\}C\{Q\} \stackrel{\text{def}}{=}$$
$$\forall \Delta_f \in \text{close}(\Lambda_f).\ (\Delta_f \models \Gamma) \Rightarrow\ \Delta_f \models \{P\}C\{Q\}$$

where

$$\Delta_f \models \Gamma \stackrel{\text{def}}{=} \forall \{P\}C.m\{Q\} \in \Gamma.\Delta_f \models \{P\}mbody(C, m)\{Q\}$$

$$\Delta_f \models \{P\}\overline{s}\{Q\} \stackrel{\text{def}}{=}$$
$$\forall S, H, I.S, H, I \models_{\Delta_f} P \Rightarrow ((S, H, \overline{s}, []) : \textbf{safe}$$
$$\wedge\ ((S, H, \overline{s}, []) \rightarrow^* (S', H', v, []) \Rightarrow S', H', I \models_{\Delta_f} Q))$$

Given this definition we can show that the two new rules for abstract predicate families are sound.

THEOREM 4.13. *Abstract weakening is sound.*

PROOF. Direct consequence of the definition of judgements and Lemma 4.9. □

THEOREM 4.14. *Abstract elimination is sound.*

PROOF. Follows from Lemmas 4.7 and 4.10 □

5. RELATED AND FUTURE WORK

In this paper we have considered the problem of writing specifications for programs that use various forms of abstraction. We have focused here on modules and Java-like classes. We have built on the formalism of separation logic and presented rules for reasoning about ADTs and Java-like classes. We have demonstrated the utility of these rules with a series of examples.

The principles of abstraction this paper builds on have been around since the Seventies. Parnas [25] first described the principles of information hiding and showed that without it seemingly independent components of a program could become tied together. Hoare provided a logic for data abstraction [11] that allowed internal implementation details to be hidden from the client. These ideas were developed further by Liskov [16] and Guttag [10] to provide what we now know as abstract datatypes.

In Hoare's [11] presentation of data abstraction, he used an abstraction function that maps values from a concrete domain to an abstract one. This abstraction function has been used in behavioural subtyping [17] to make classes with different implementations meet the same specification. When reasoning with framing, Leino observed that, in addition to abstraction functions, datagroups [14] are needed to abstract *modifies*[14] clauses. Abstract predicates, and families, combine the concept of both datagroups and the abstraction function into a single definition: separation logic formulae represent both the amount of state and its possible values.

There have been several attempts to reason about Java using a Hoare logic, including those by Oheimb and Nipkow [24], Poetzsch-Heffter and Müller [27], Pierik and de Boer [26]. However these logics do not have the framing properties of separation logic; method bodies must be verified at each call site. Also they do not attempt to express abstraction. In this sense Leino's work with datagroups [14] and data abstraction [15] are more closely related. This work uses the concept of "modular soundness" to determine when state can be exposed to a client. Another related approach by Barnet et al. [2] uses a private invariant to encapsulate the objects state. This invariant can be "packed" and "unpacked" to access its contents. These pack and unpack operations can be seen as corresponding to the open and close implications of abstract predicates.

Reddy [28] takes a different approach to adding abstraction to the logic. He extends specification logic to provide the ability to existentially quantify a predicate. This quantified predicate behaves in a similar way to an abstract predicate.

Middelkoop et al. [18] have similar aims to us and give a separation logic for a class-based language. Their approach considers an object as the primitive element of the heap, rather than a field. This restricts their use of the frame rule by preventing them from considering splitting an object. Their work does not consider abstraction or inheritance and so can not handle any of the examples presented in this paper.

A different approach to adding abstraction to separation logic has been taken by O'Hearn et al. [22]. They use the hypothetical frame rule to reason about static modularity. They are not able to reason about ADTs or classes, and cannot verify the examples we present. All the examples they present can by expressed using abstract predicates, however the proofs are less compact: predicates must be threaded through the proof to represent the internal invariant. This leads to two open questions: (1) can abstract predicates express all the proofs of the hypothetical frame rule?; and (2) can the concepts be soundly combined into a single logic? We believe the answer to both of these questions to be yes, but more work remains.

Building on the principles of the hypothetical frame rule, Mijajlović and Torp-Smith [19] have built a semantic model of refinement in a setting similar to separation logic. This allows them to semantically show one module could be used in place of another. They do not provide any logical rules for this reasoning, and they do not deal with ADTs. It would be interesting to see if their models could be adapted to abstract predicates.

A different approach to separation logic to dealing with the problem of aliasing is to impose some form of restriction using a type system. Ownership types [6] have been used to restrict aliasing in object-oriented languages. They prevent pointers into an object's representation, which helps reasoning about encapsulation. Smith and Drossopoulou [7] exploit this encapsulation to extend a Hoare logic with framing properties. Separation logic is more flexible than ownership types as it prevents dereferencing of a pointer rather than its existence.

Many researchers have pointed to similarities between separation logic and ownership types. However close comparison has been hampered by the fact that separation logic research has dealt with

[14] A *modifies* clause is an annotation that specifies all the possible changes made by a method/function body.

low-level pointer manipulation; whereas ownership types has dealt with high-level object languages. We hope that the work detailed in this paper may provide a stepping-stone for a more indepth analysis of these two approaches.

In this paper we have built a logic for reasoning about abstract types. It is well-known that abstract types correspond via the Curry-Howard correspondence to existential types [20]. Abstract predicates appear to be analogous, but at the level of the propositions themselves. We should also like to explore this analogy further, perhaps using higher-order logic to provide a logical semantics for abstract predicates.

The types analogy leads to another direction to pursue: parametric polymorpism. In this paper functions and methods can be defined to manipulate a datatype or class without knowing its representation: e.g. the connection pool did not know how a connection was stored. This is related to O'Hearn's comment that "Ownership is in the eye of the asserter". Abstract predicates may provide a suitable setting for studying parametric datatypes; we are currently working on a set of proof rules.

Finally, in this paper we have only considered sequential languages. Recently, O'Hearn has shown how to extend separation logic with rules to reason about concurrency primitives [21]. These rules use the same information hiding principles of the hypothetical frame rule [22]. They allow state to be stored in a semaphore, and by manipulating this semaphore the state can be transfered between threads. Unfortunately the semaphore is statically scoped, which prevents reasoning about heap allocated semaphores including, for example Java's `synchronised` primitive. We are currently consider the combination of the information hiding provided by abstract predicates with O'Hearn's system for concurrency to allow for reasoning about semaphores in the heap, and hence Java with threads.

Acknowledgements

We should like to thank Peter O'Hearn for insightful comments on earlier versions of this work and proposing the malloc and free example; and Andrew Pitts, Alisdair Wren and the anonymous referees for comments on this paper. We acknowledge funding from EPSRC (Parkinson) and APPSEM II (Bierman and Parkinson).

6. REFERENCES

[1] M. Abadi and L. Cardelli. *A theory of objects*. Springer, 1996.

[2] M. Barnett, R. DeLine, M. Fähndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 2004.

[3] G.M. Bierman and M.J. Parkinson. Effects and effect inference for a core Java calculus. In *Proceedings of WOOD*, volume 82 of *ENTCS*, 2004.

[4] L. Birkedal, N. Torp-Smith, and J.C. Reynolds. Local reasoning about a copying garbage collector. In *Proceedings of POPL*, 2004.

[5] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permissions accounting in separation logic. Proceedings of POPL, 2005.

[6] D. Clarke and S. Drossopolou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of OOPSLA*, 2002.

[7] S. Drossopoulou and M. Smith. Cheaper reasoning with ownership types. In *Proceedings of IWACO*, 2003.

[8] J. Ellis and L. Ho. JDBC 3.0 specification, 2001. `http://java.sun.com/products/jdbc/download.html`.

[9] M. Grand. *Patterns in Java*, volume 1. Wiley, second edition, 2002.

[10] J. Guttag. *The Specification and Applications to Programming of Abstract Data Types.* PhD thesis, Dept. of Computer Science, University of Toronto, 1975.

[11] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[12] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[13] J. Lamping. Typing the specialization interface. In *Proceedings of OOPSLA*, 1993.

[14] K.R.M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of OOPSLA*, 1998.

[15] K.R.M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24:491–553, September 2002.

[16] B. Liskov and S.N. Zilles. Programming with abstract data types. In *Proceedings of Symposium on Very High Level Programming Languages*, 1974.

[17] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, 1994.

[18] R. Middelkoop, K. Huizing, and R. Kuiper. A Separation Logic Proof System for a Class-based Language. In *Proceedings of LRPP*, 2004.

[19] I. Mijajlović and N. Torp-Smith. Refinement in a separation context. In *Proceedings of FSTTCS*, 2004.

[20] J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

[21] P.W. O'Hearn. Resources, concurrency and local reasoning. Invited paper, in *Proceedings of CONCUR*, 2004.

[22] P.W. O'Hearn, H.Yang, and J.C. Reynolds. Separation and information hiding. In *Proceedings of POPL*, 2004.

[23] P.W. O'Hearn, J.C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, 2001.

[24] D. Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In *Formal Methods Europe*, 2002.

[25] D.L. Parnas. The secret history of information hiding. In *Software Pioneers: Contributions to Software Engineering*. Springer, 2002.

[26] C. Pierik and F.S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In *Formal Methods for Open Object-Based Distributed Systems*, 2003.

[27] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In *Proceedings of ESOP*, 1999.

[28] U.S. Reddy. Objects and classes in Algol-like languages. *Information and Computation*, 2002.

[29] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, 2002.

[30] R. Stata. Modularity in the presence of subclassing. Technical Report 145, Digital Equipment Corporation Systems Research Center, April 1997.

[31] H. Yang. *Local reasoning for stateful programs*. PhD thesis, University of Illinois, July 2001.