
© Copyright 2008 QUT & Andrew Craik and Wayne Kelly
ABSTRACT
The principal premise of this paper is that as a field, we do not currently have a suitable conceptual framework for reasoning about inherent parallelism. We have techniques called dependence analysis for determining the safety of parallelization, but these techniques do not provide a basis for abstraction and so do not scale well to entire applications that are large, complex and developed from components. This paper seeks to extend both the principles and practice of parallel programming by proposing a new abstraction for reasoning about inherent parallelism and uses that abstraction to develop a set of sufficient conditions for data parallel loops.

1. INTRODUCTION
This paper addresses the problem of identifying sections of a program that can be safely executed in parallel. Despite decades of attempts, such automatic parallelization is still, in general, well beyond the state-of-the-art. The challenge faced by programmers trying to manually parallelize a program is similarly daunting for the same reasons. With the emergence of multi-cores, parallelism has now entered the mainstream; failure is no longer an option. Given the difficulty of the problem, we must find a way to reformulate it so that it becomes more tractable. We seek a solution that is simple, yet yields sufficient conditions for parallelism that are permissive enough to be useful.

We believe the problems with traditional approaches are two fold: (1) they are too fine grained and (2) they do not facilitate abstraction. Traditional analyses deal with complex array index expressions and pointer may-alias questions; the analysis is performed at the level of individual memory locations. These problems are known to be undecidable in the general case. Further, traditional dependence analysis provides no basis for abstraction - when we analyze code containing a function or method call, the function signature provides no useful information about possible inter-procedural data dependencies. We are forced to examine the implementation of the function and all functions that it might call. Inter-procedural analysis simply does not scale well to large, complex applications. The pervasive use of dynamic linking and late binding in modern component-oriented software systems only further complicates the situation making whole program analysis not generally possible.

We address the problem of reasoning about inherent parallelism in the specific context of imperative object-oriented languages. We do this for two reasons. Firstly, the emergence of multi-cores means that parallelism will now enter the domain of general purpose desktop and sever applications and object-orientation is the paradigm ‘du jour’ for developing such applications. Secondly, object-oriented programming provides structure to the memory allocated by the program and we seek to exploit this structure to facilitate reasoning at higher levels of granularity.

We borrow concepts from the field of Ownership Types [3] and apply them to the problem of reasoning about inherent parallelism. Specifically, we use ownership contexts to abstract the side-effects of methods by listing the contexts a method may read or write as part of its signature. We then use this abstraction to construct sufficient conditions for the safe parallelization of data parallel loops as well as for task style parallelism. Our abstractions provide a system for reasoning about parallelism that is intended for use by either programmers or automated tools. This paper does not address the questions of which parallelism should be exploited; we seek only to reason about what inherent parallelism exists.

Our specific contributions in this paper are:

- A conceptual framework for abstracting and reasoning about side effects and data dependencies in imperative object-oriented languages (Section 2.3)
- Sufficient conditions for the safe parallelization of data parallel loops (Section 3) and task style parallelism (Section 6)

2. BACKGROUND
We are certainly not the first researchers to propose exploiting the structure of object-oriented programs to facilitate more abstract reasoning about program properties, but there has been relatively little work using these techniques for parallelization analysis. This is partially due to the fact that, to date, parallelization work has largely focused on scientific applications not generally written in object-oriented languages.
2.1 Object-Orientation

Object-oriented programming languages provide more structure to the memory allocated by a program compared, for example to, C or Fortran. Specifically, data is grouped into objects which in turn form a hierarchy of representation due to the concept of encapsulation (the idea that some objects are considered part of the internal representation of other objects). This structure can be exploited to facilitate reasoning about disjointness of reference.

2.2 Ownership Types

Consider the following code snippet:

```java
private Object[] signers;
...
Object[] getSigners() {...};
```

Note that despite the private annotation on the signers field, it is possible for the `getSigners` method to return a reference to the object referenced by the private field; the private annotation only protects the name of the field and not the data contained in the field. This code was the source of the infamous `getSigners` bug in Java 1.1.1 precisely because the data in the `signers` field was not protected by the field annotations. Providing this type of protection in a rigorous manner is generally called strong encapsulation enforcement and a number of different systems to enforce it have been proposed. We have chosen to initially base our parallelization work on one of these systems called Ownership Types [3].

Enforcing encapsulation requires each object to track: (1) which object’s representation it is part of and (2) which objects are part of its representation. Clarke, Noble, and Potter abstracted this tracking through the notions of ownership and object contexts (hereafter referred to as contexts) [3]. As they eloquently described it, ‘Each object owns a context, and is owned by the context that it resides within’ [3]. Top-level objects not part of any objects’ representation are said to be owned by the special top context `world`. The context owned by an object is unique and is referred to by the object as its `this` context. The ownership structure created by these definitions produces a forest of tree-ordered contexts rooted in the `world` context. Encapsulation is enforced in these systems by preventing the escape of references outside of their own contexts. The principle mechanism employed to prevent the escape of such references is type naming restrictions; there is literally no way for an outer class to express the name of a hidden inner context as only the class itself can name its `this` context.

Our system is based on an extended version of Clarke, Potter and Noble’s original ownership types approach [3]. We borrow the basic notion of ownership contexts from these earlier works; however, we have very different goals. Those works are concerned with enforcing strong encapsulation in order to argue about correctness properties of programs. We have no particular interest in enforcing strong encapsulation, we seek only to accurately track which contexts are read and written. We are, therefore, able to relax many of the rules dictated by these earlier systems. We are less judgmental about who can write to particular contexts and as a consequence it becomes much easier for programmers to annotate their programs as “our” type rules are strictly weaker.

For a practical demonstration of how ownership is commonly expressed in a program, consider the following code snippet:

```java
class Link[o,d] {
    Link[o,d] next;
    Data<d> dt;...
}
```

Class declarations add a list of formal context parameters after the class name in square brackets. While the syntax superficially resembles that used for generic types, in this case the parameters represent not types, but contexts corresponding to actual objects. By convention, the first of these parameters represents the context of the objects that will own instances of this class. Other formal context parameters may also be passed to be used inside the class in the construction of other concrete ownership types. Concrete ownership types are constructed by naming a class parameterizing it with an appropriate number of actual context parameters listed inside angle brackets.

2.3 Ownership Relationships for Reasoning

Now that we have the concept of ownership contexts, it is possible to abstract the operation of a method in terms of the affected contexts. We simply augment method signatures with the set of contexts directly or indirectly read and written by that method.

For example, consider the following code snippet:

```java
class DataHolder[o,d] {
    private Data<d> heldData;
    Data<d> getHeldData() reads<this> writes<> {
        return heldData;
    }
}
```

The `getHeldData` method above simply returns the value of a field. To obtain the value of the field we must read the data in the current object, that is data in the `this` context. This read appears on the effect list as does an empty write effect list since no contexts are written by the method.

Consistency between the computed and declared effects can be compiler enforced, see Section 7. The only time that declared effects are strictly necessary is when calling methods in other software components where the source is not available. In all other cases, the compiler can infer the effects although it may still be useful to explicitly declare effects in order to formally record design intentions or to facilitate overriding.

Other researchers have proposed similar mechanisms for abstractly capturing the side-effects of method invocations; however, they were not explicitly designed for, nor applied to, the problem of reasoning about parallelism. For example, Greenhouse and Boyland presented an abstract effects system for Java [6] and Smith proposed an abstract effect system for ownership domains [12].

2.3.1 Abstracting Effects

A method’s effect signature must include the effects of all the methods it might call. We cannot, in general, guarantee that a method caller is able to name all of the contexts nameable by the methods it calls. Allowing the caller to do so would violate encapsulation and expose implementation details to callers. We solve this problem by noting that because contexts are nested, effects can be generalized. Reading an object’s `this` context is included in reading its
dependencies exist: conditions which are sufficient to ensure that none of these anti-dependencies. We now state some informal sufficient dependencies can take one of three forms: output, flow, and ensure that there are no inter-iteration dependencies. These for the above loop to execute safely in parallel we must 3. DATA PARALLELISM

Now consider the following stereotypical data parallel loop:

```java
foreach(T c> element in collection) {
    element.operation(arguments);
}
```

For the above loop to execute safely in parallel we must ensure that there are no inter-iteration dependencies. These dependencies can take one of three forms: output, flow, and anti-dependencies. We now state some informal sufficient conditions which are sufficient to ensure that none of these dependencies exist:

1. The elements of the collection must be unique; no two elements refer to the same object.
2. The operation only mutates its “own” element’s state and does not read the state of any of the other elements.

Element uniqueness is assumed by many parallelization algorithms; Cohen, Wu, and Padua coined the term “comb” to succinctly describe collections as containing unique items (an analogy to the parallel pointing teeth of a comb) [4]. In order to parallelize such loops, we assume that the type of the collection is somehow annotated as having this comb property. Adherence to this comb property can generally not be determined via static analysis. We could either alter the collection class to test and assert this comb invariant or alternatively, to avoid this runtime overhead, we could simply trust the programmer to respect this declared invariant.

Condition 2 says that the write set of the operation contains only this, or is the empty set. The read set can contain this, but it may also contain other contexts r, provided that we know that r is disjoint from the contexts written by the operation when it is applied to the other elements in the collection.

If condition 1 is not known to be true or if the write set of the operation contains any context other than this then it does not meet our definition of a data parallel loop. We have no basis on which to assume such loops can be executed safely in parallel and so will conservatively indicate that they should be executed sequentially.

If condition 1 is true and the write set of the operation is empty then we can say without doubt that the loop can be parallelized. Similarly if the write set of the operation is this and the read set is this or empty then the loop can definitely be parallelized.

The more interesting case is where the operation also reads other shared state belonging to various other contexts r. In this case, the answer is maybe - the loop is parallel provided that all of the read contexts r are provably disjoint from the collection. As we shall argue, proving disjointness of such contexts if very difficult to do statically. In general, we must resort to a runtime test of disjointness that allows us to generate conditionally parallel loops:

```java
if (/*runtime test: r disjoint from collection*/)  
    parallel foreach(element in collection)  
        element.operation(arguments);  
else  
    serial foreach(element in collection)  
        element.operation(arguments);  
```

3.1 Reading Shared Mutable State

Figure 1 illustrates abstractly what we would like to be able to support: a data parallel loop reading from shared mutable state. We have a loop that iterates over elements d₁,…,dₙ of some collection owned by object c. The operation invoked on those elements writes only to its own element, but also reads objects owned by some context r elsewhere in the ownership tree. In this example, it is safe to parallelize the loop as r can be seen to be disjoint from c since they are on different branches of the ownership tree.

Figure 1: ownership relationships between contexts at runtime used for example of capturing context disjointness

The ownership relation defines a partial order on the object contexts. Two arbitrary contexts might be either:

- equal (=), they are one and the same

```java
class DataHolder[o,d] {
    private Data<d> dt;
    Data<d> getData() reads<this> writes<> {
        return dt;
    }
}

class DataWrapper[o,d] {
    private DataHolder[o,d> holder;
    Data<d> getData() reads<this, o> writes<> {
        return holder.getData();
    }
}
```

Note that the read effect of getData is reading the this of DataHolder. We cannot name the holder’s this context within the DataWrapper. We note that the owning context of the DataHolder is o so we can abstract the read as a read of o giving us the final effect set of this and o.

This combination of contexts and abstraction provides the basic framework for describing effects which facilitates reasoning about parallelism and stating sufficient conditions for parallelization to be safe.

The ownership relation defines a partial order on the object’s this context. This generalization is necessary because in summarizing the effect of a method we can only make use of contexts that are nameable by callers of that method. Contexts which are only visible within the implementation of the method must be abstracted, or raised up and summarized, as the closest containing context nameable from the viewing scope. We call this operation raising and it is formally defined as part of our type system in Section 7.2.1.

To demonstrate the raising operation consider the following contexts. Two arbitrary contexts might be either:

- equal (=), they are one and the same
• ordered (<), one is an ancestor of the other

• unordered (<>), they appear on different branches of the tree

So, our sufficient condition can be formally stated as:

• The collection is declared to have the comb property

• the operation writes to at most this

• the operation reads only from this and other contexts \( r \) such that \( r <> c \)

4. RUNTIME REPRESENTATION

We wish to allow data parallel loops to read from virtually any object in the entire program. The only objects they are not allowed to read from are objects contained within the collection being iterated over. The references to these objects that are read and written can be obtained in many ways: they can be passed in as parameters (possibly from much higher abstraction layers) or they can be obtained by accessing fields, properties or calling methods of other objects that we can reference. It is possible to statically track the owner of each object; however, it is very difficult to statically track the ownership relationships between all possible combinations of objects that we might read or write. Therefore, in general, we argue it is necessary to resort to some form of runtime mechanism and test to determine the ownership relationships of arbitrary objects.

A naïve runtime representation of ownership relationships would simply require each object to maintain a special hidden reference to the object that owns it. Such a straight forward tree data structure could easily be used to determine the relationship between two arbitrary objects; however, it might take \( O(n) \) time to perform such comparisons in the worse case. Fortunately, the problem of determining Nearest Common Ancestors (NCA) in trees has been well studied and there are clever data structures and algorithms based on node numbering and other techniques that use only \( O(n) \) memory to represent a tree containing \( n \) nodes and can determine if two nodes are ordered in \( O(1) \) time [11].

5. FACILITATING UPWARD DATA ACCESS

So far we have formulated a sufficient condition for data parallel loops designed to allow restricted reading of shared mutable state and we have a runtime mechanism to assist in testing this condition. Unfortunately, this is not sufficient to achieve our goal. Consider again the example shown in Figure 2. It is clear that a data parallel loop (in a method executing at or below context \( c \)) can safely read data from context \( r \). Unfortunately, it cannot name context \( r \), so context \( r \) cannot appear in its read set; rather, it would be generalized as a read of \( b \). But if all we know is that it may read from \( b \) then this might include anything under \( b \) including the collection itself making it unsafe to parallelize code containing such effects.

Figure 2: ownership relationships between contexts at runtime used for example of capturing context disjointness

If somehow, \( r \) was in the read set of the operation, then we could use our runtime mechanism to test that it was unordered with respect to \( c \) and so safe to parallelize. We, therefore, require some mechanism that will result in the read of \( r \) not being generalized as a read of \( b \).

We add two features to our system that both achieve this effect: method level context parameters and sub-contexts. Firstly, we allow methods to introduce their own formal context parameters that are independent of the containing classes’ formal context parameters. Such method level context parameters are bound when the method is called. These parameters allow a third party (who has references to both \( r \) and \( c \)) to make a method call that passes a parameter we can read from that is known to be owned by context \( r \). Note that this modification to the ownership type system is only possible because (1) we are not concerned with enforcing encapsulation, we are only concerned about accurately tracking which contexts are read and written, and (2) we have a runtime mechanism that allows us to determine the ownership relationship of \( r \) and \( c \).

If we are not passed a parameter owned by context \( r \), then the only way we can access such state is via a method or field of object \( b \), the least common ancestor of \( r \) and \( c \). When making such read accesses we want to be able to reason that the part of \( b \) that we are accessing is disjoint from the part of \( b \) that contains the collection. We therefore introduce the notion of sub-contexts to allow us to partition the contents of objects like \( b \). So, object \( b \) would still own \( b_1 \) and \( b_2 \), but they could be placed into separate sub-contexts.

We only permit the this context and the world context to contain sub-contexts. Using these sub-contexts, reading \( r \) could be summarized as the named sub-context of \( b \) containing \( b_1 \) rather than \( b \) itself. If the collection is located in sub-context \( b_2 \), then we could safely allow a read of data in sub-context \( b_1 \) since such a read could never read the collection being traversed.

For each class, the programmer can decide if they wish to declare sub-contexts and if they do, they can declare as many as they desire. In the extreme case, each private field might be given its own sub-context, but programmers would more commonly create a sub-context to encapsulate a group of related private fields. The more sub-contexts, the more information that needs to be passed as actual context arguments on types; the creation of sub-contexts is a trade-off between precision and complexity. Sub-contexts are limited in scope to their class of declaration, to children they look...
like any other context passed down from the parent while to parents they appear to be part of the owning class’ representation. This limits the scope of changes required to introduce these sub-contexts to the class scope.

6. TASK PARALLELISM

Given that we know how to test if a loop is parallel, it is a relatively straightforward extension to derive sufficient conditions for two sections of code to execute in parallel (task style parallelism). We simply compute the total effect of both code sections and ensure that the relationship between their read and write sets is such that no flow, output or anti-dependences exist.

Considering ownership alone is simple and clearly provides sufficient conditions for parallelization, however, consider for example deciding if the statement:

\texttt{o1.method1();}

can be executed in parallel with:

\texttt{o2.method2();}

If \texttt{o1} and \texttt{o2} are both owned by the same object, then our sufficient conditions may tell us that they are not parallelizable. However, if we knew via a runtime test that \texttt{o1} was not equal to \texttt{o2} then this may provide enough information to conclude that they are parallelizable. Exploring this tradeoff between a more accurate test and the complexity of testing will be a topic of future work.

7. TYPE RULES

Now that we have described the salient features of our system in an informal manner, we proceed to develop the formal specification of a simplified toy language demonstrating these features. This formalization will be utilized in the formal proof of our sufficient conditions for loop parallelism in Section 8.

7.1 Abstract Syntax

A program \( P \) is defined to be a set of classes \( L \) and static boot-strapping expression \( e \):

\[ P ::= L e \]

The definition of a class \( C_1 \) with formal context parameters \( \bar{X}_1 \) which optionally extends a class \( C_2 \) consists of a set of sub-contexts \( \bar{X}_3 \), a set of fields \( T \) with types \( T \) and a set of method declarations \( M \):

\[ L ::= \text{class } C_1 \{ \bar{X}_1 \} \text{ extends } C_2 \{ \text{subcontexts } \bar{X}_3; \{ T; M \} \}
\]

A type \( T \) consists of the name of a class \( C \) and a set of actual context parameters \( \bar{K} \):

\[ T ::= C(\bar{K}) \]

The declaration of a method with return type \( T \) named \( m \) with formal context parameters \( X \) taking parameters \( \bar{x} \) of types \( T \) with maximum read effects of \( T \) and maximum write effects of \( T \):

\[ M ::= Tm[X] \{ Tx \} \text{reads}(T) \text{writes}(T) \{ \bar{x} \} \]

Expressions evaluate to values and consist of method invocations, object instantiations, use of formal parameters, field reads, and references to this and super:

\[
  e ::= e.m(\bar{K})(\bar{x}) | e.f \\
  | \text{new } C(\bar{K})(\bar{x}) | \text{this} \\
  | \text{super} \\
\]

A statement consists of an expression, assignment, sequence of statements, a return, or a foreach loop:

\[
  s ::= ; | e; \\
  | e1.f = e2; | \{ \bar{x} \} \\
  | \text{return } e; | \text{foreach } (Tx \in e) \{ \bar{x} \}
\]

Actual context parameters can be:

\[
  K, I, J ::= X|this.X|world.X
\]

\( \varphi \) is a tuple of read effects \( I \) and write effects \( J \):

\[
  \varphi ::= \langle I, J \rangle
\]

Type checking takes place in an environment \( \Gamma \) which holds mappings from variables to types as well as domination relationships between contexts:

\[
  \Gamma \in \{ x \mapsto T, \text{variable} \}
  \\
  K \preceq \bar{K}, \text{domination} \\
  K \rightarrow T \}
\]

contexts mapped to referring classes

Lastly, we track the current method being typed, as specified by its name and parameters, in a method frame \( \Delta \):

\[
  \Delta ::= \langle m, T \rangle
  \\
  | \emptyset
\]

7.2 Helper Functions

There are a number of helper functions which we use to lookup information about methods, fields, and classes. The method function returns the return type, read and write effects, and formal context arguments of a method \( m \) in class \( C \) with arguments of type \( T \):

\[
  \text{class } C[X_1] \ldots [X_m] \{ Tm[X_t] \langle T \rangle \text{reads}(T) \text{writes}(T) \ldots \}
  \\
  \varphi = \langle T, J \rangle
  \\
  \text{method}(C(\bar{K}_1), m, T) = \langle [K_1/X_1]T; \bar{K}_1/X_1 \varphi, \bar{X}_1 \rangle
\]

\[
  \text{class } C \ldots \text{extends class } C'[\ldots M \ldots]
  \\
  \text{class } C[X] \quad m(\bar{M}) \notin \bar{M} \quad K_2 = K_1[\bar{X}_1]
  \\
  \text{method}(C(\bar{K}_1), m, T) = \text{method}(C'(\bar{K}_2), m, T)
\]

\[
  \text{method}(C(\bar{K}), m, T) = \emptyset
\]

The field method returns the type of a field \( f \) in a class \( C \):

\[
  \text{class } C[X] \ldots [\ldots Tf \ldots]
  \\
  \text{field}(C(\bar{K}), f) = [\bar{K}/X]T
\]
The subcontexts function returns the declared sub-contexts of the this context in class C:

\[ \text{class } C \ldots \text{extends } C' \{ \ldots \} \]
\[ \text{class } C'[\{ X \} \quad f \notin \mathcal{E}] \]
\[ \text{field}(C'\langle K \rangle, f) = \text{field}(C'\langle K_{1..}X \rangle, f) \]

\[ \text{class } C \{ \ldots \mathcal{E} \ldots \} \quad f \notin \mathcal{E} \]
\[ \text{field}(C\langle K \rangle, f) = \emptyset \]

The special contexts world and its sub-contexts are globally visible and so do not change:

\[ \text{raise}(\text{world}, \omega) = \text{world} \]
\[ \text{raise}(\text{world}, X, \omega) = \text{world}X \]

### 7.3 Type Rules for the Language

In the following subsections the standard format of the typing statements will be:

\[ \Gamma; K; \Delta \vdash e : \Gamma \]

This statement is read as the expression \( e \) evaluates to type \( \Gamma \) with side-effects \( \varphi \) under typing environment \( \Gamma \) with current context \( K \) and current method frame \( \Delta \).

#### 7.3.1 Programs

To type a program we validate all of the classes defined in it and then type the bootstrap code and compute the program’s return type and effects based on it:

\[ \vdash L \quad \varnothing; \text{world}; \varnothing \vdash e : \Gamma \]
\[ Le : \pi \Gamma \]

#### 7.3.2 Class Declarations

To validate a class declaration, we must ensure that the class it extends, if any, is valid, that fields are not overridden, and that the declared formal context parameters only append additional parameters to the list declared by the super class. Note that the super class is optional since our type system does not require a top type.

\[ \Gamma = \text{this} \leq X_1, \text{this} : C_1(\langle X_2, X \rangle), \text{this} : C(\langle X \rangle), \text{super} : C_2(\langle X' \rangle), \text{this} : C(\langle X \rangle) \]
\[ \forall f \in \mathcal{E} \quad \text{field}(C_2(\langle X' \rangle), f) = \emptyset \quad \forall i \in 0..|X'| \quad X_i = X_i' \]
\[ \text{class } C_2(\langle X' \rangle) \ldots \quad \Gamma; \text{this} \vdash \text{class } C_2(\langle X' \rangle), T' \]
\[ \Gamma; \text{this} \vdash M \]
\[ \vdash \text{class } C_1 \text{ extends } C_2 \{ \text{subcontexts } X'; T'; M \} \]

\[ \Gamma = \text{this} \leq X_1, \text{this} : C(\langle X_2, X \rangle), K : C(\langle X \rangle), \text{this} : C(\langle X \rangle) \]
\[ \forall i \in 0..|X| \quad X_i = X_i' \quad \Gamma; \text{this} \vdash T \]
\[ \vdash \text{class } C \{ \text{subcontexts } X'; T'; M \} \]
7.3.3 Method Definition

To validate a method definition, we first type its constituent statements in the current evaluation environment with the formal parameters bound to their type to determine the effect of executing the method body. The computed effects must be the same or smaller than the effects declared on the signature. Further, the declared effects must be the same or smaller than those of the method being overridden, if any. Lastly, the method must include its parent's formal context parameters, but may optionally add its own parameters as well (validated by the $\forall i \in 1..|\mathcal{X}'| \ X_i' = X_i$ below).

$$\Gamma, \pi : \mathcal{T}; K; \{ \langle m, \mathcal{T} \rangle \} : \{ \mathcal{T}, \mathcal{F} \} \emptyset$$

$$\text{method}(\Gamma(\text{super}), m) = T m[\mathcal{X}](\mathcal{T}_-) \text{reads}(\mathcal{T}^m)\text{writes}(\mathcal{F}^m)$$

$$\Rightarrow \mathcal{T} \leq \mathcal{T}' \land \mathcal{F} \leq \mathcal{F}' \land \forall i \in 1..|\mathcal{X}'| \ X_i' = X_i,$$  

$$\Gamma; K; \emptyset \vdash \text{next}() \text{reads}(\mathcal{T}) \text{writes}(\mathcal{F})\{\pi\}$$

$$\Gamma, \pi : \mathcal{T}; K; \{ \langle m, \mathcal{T} \rangle \} : \{ \mathcal{T}, \mathcal{F} \} \emptyset$$

$$\text{super} = \emptyset \lor \text{method}(\Gamma(\text{super}), m) = \emptyset$$

$$\Rightarrow \mathcal{T} \leq \mathcal{T} \land \mathcal{F} \leq \mathcal{F}.$$  

$$\Gamma; K; \emptyset \vdash \text{next}(\mathcal{T}x \in e)\{\pi\} : \emptyset$$

7.3.4 Constructor Definition

Constructor definitions are typed in a similar manner as method definitions. Because the object being initialized only becomes accessible once the constructor returns, read and write effects of this can be safely removed from the constructor's effects.

7.3.5 Loops

The foreach loop considered earlier in this paper can be typed in this system. We require the collection in the loop to have a $\text{next}()$ method which returns an object with a type which is included in the declared element type:

$$\Gamma; K; \Delta \vdash e : \varphi' C(\mathcal{K}) \quad \text{class } C[\mathcal{X}] \ldots$$

$$\text{method}(C(\mathcal{K}), \text{next}, \emptyset) = \langle T', \varphi', \varnothing \rangle$$

$$\varphi = \varphi' \cup \varphi'' \quad T' < : T \quad \Gamma, e \rightarrow T; K; \Delta \vdash \{ \pi \} : \varphi'' \emptyset$$

$$\Gamma; K; \Delta \vdash \text{foeach } (\mathcal{T}x \in e)\{\pi\} : \emptyset$$

7.3.6 Statement Blocks and Expressions

To type a block of statements we simply type each of the statements; there is no result type because statements only produce side-effects:

$$\forall s_i \in \pi; \Gamma; K; \Delta \vdash s_i : \varphi_i \emptyset \land \varphi = \bigcup \text{raise}(\varphi_i, K, T_i)$$

$$\Gamma; K; \Delta \vdash \{ \pi \} : \emptyset$$

When typing an expression as a statement, we discard the result type:

$$\Gamma; K; \Delta \vdash e : \emptyset T$$

$$\Gamma; K; \Delta \vdash e : \emptyset \emptyset$$

7.3.7 Return Statements

To type a return statement, we must ensure that the type of the expression to be returned is a valid subtype of the current method’s return type. Finally, the effect of evaluating the return is the effect of evaluating the expression to be returned.

$$\Gamma \vdash K : C(\mathcal{K}) \quad \Gamma; K; \{ \langle m, T \rangle \} \vdash e : \emptyset T'$$

$$\text{method}(C(\mathcal{K}), m, T) = \langle T', \varphi, \emptyset \rangle \quad T' < : T$$

$$\Gamma; K; \{ \langle m, T \rangle \} \vdash \text{return } e : \emptyset \emptyset$$

7.3.8 Method Invocation

To type a method invocation we first compute the type and effect of evaluating the expression $e$. We can then compute the types and effects of computing the method’s actual parameters. We then lookup the size of the method’s context parameter list and ensure a valid actual context parameter has been supplied for each. The effect of the invocation is the union of these read-effects and write-effects combined with the method’s declared effects raised to the current context after substituting actual contexts for formal context parameters.

$$\Gamma; K; \Delta \vdash e : \emptyset' C(\mathcal{K}_2) \quad \Gamma; K; \Delta \vdash \pi : \emptyset T \quad \text{method}(C(\mathcal{K}_2), m, T) = \langle T, \varphi, \emptyset \rangle$$

$$\varphi \text{ union } \varphi' \quad \varphi = \text{raise}(\varphi', K, C(\mathcal{K}_2)) \cup \text{raise}(\varphi, K, T)$$

$$\text{method}(\langle \mathcal{K}_1 / \mathcal{X}_1 \rangle \varphi, K, C(\mathcal{K}_2))$$

$$\Gamma; K; \Delta \vdash e.\text{m}(\langle \mathcal{K}_1 \rangle(\pi)) : \emptyset' T$$

7.3.9 Object Instantiation

Calling a constructor is largely the same as calling a method except that for simplicity there are no formal context parameters to bind and there is no receiver computation required. Note that the type of the object being created is validated to ensure that the correct number of context parameters are supplied.

$$\Gamma; K; \Delta \vdash \pi : \emptyset T \quad \Gamma; K; \Delta \vdash C(\mathcal{K}) \quad \text{method}(C(\mathcal{K}), C, T) = \langle \varphi, \emptyset', \emptyset \rangle \quad \text{class } C[\mathcal{X}] \ldots$$

$$\varphi' = \text{raise}(\pi, K, T) \cup \text{raise}(\mathcal{K}/\mathcal{X}, \varphi, K, C(\mathcal{K}))$$

$$\Gamma; K; \Delta \vdash \text{new } C(\mathcal{K})(\pi) : \emptyset' C(\mathcal{K})$$

7.3.10 Formal Parameters

There are no primitive types or local variables, so reading a variable is simply reading a value that is a reference to an object. Reading an argument does not, therefore, read or write the state of any objects:

$$\Gamma(x) = T$$

$$\Gamma; \omega \vdash x : (\emptyset, \emptyset') T$$

Reading the local self-reference variable this has no side-effects for the same reasons:

$$\Gamma(x) = T$$

$$\Gamma; \omega \vdash \text{this} : (\emptyset, \emptyset') T$$

$$\Gamma; \omega \vdash \text{super} : (\emptyset, \emptyset') T$$


7.3.11 Reading Fields

When reading a field, we must first compute the type of the object to which the field belongs. The effect of the statement will then be the total read and write effects of evaluating the object reference expression as well as a read of the context or sub-context in which the field is located. However, we must raise the context of computing the object reference up to a level of abstraction that can be named from within the current class' context $K$.

\[
\Gamma; K; \Delta \vdash e : \phi \cdot C(\overline{K}) \quad \text{field}(C, f) = T \\
\text{if}(\text{owner}(T) < K_1) \text{ then } Y = \text{owner}(T) \text{ else } Y = K_1 \\
\varphi = \text{raise}(\phi', K, C(\overline{K})) \cup \{Y, \emptyset\} \\
\Gamma; K; \Delta \vdash e : \phi \cdot T
\]

7.3.12 Writing Fields

To compute the effect of writing to a field we must compute the types and effects of evaluating the object reference expression and the new value for the field. These effects are then raised to the current context and the owner of the field's object is added to the write effects:

\[
\Gamma; K; \Delta \vdash e : (T, T') \cdot C(\overline{K}) \quad \Gamma; K; \Delta \vdash e' : (T, T') \cdot T' \\
\text{if}(\text{owner}(T) < K_1) \text{ then } Y = \text{owner}(T) \text{ else } Y = K_1 \\
\varphi = \left\{ \begin{array}{l}
\text{raise}(T, K, C(\overline{K})) \cup \text{raise}(T, K, T') \\
\text{raise}(T, K, C(\overline{K})) \cup \text{raise}(T, K, T') \cup \{Y\}
\end{array} \right. \\
\Gamma; K; \Delta \vdash e : \phi' \cdot \emptyset
\]

7.4 Validating Contexts and Types

Lastly, we present rules for validating contexts and types. These are similar to more recent ownership types systems like that proposed by Lu and Potter [7] due to the addition of method-level context parameters and the removal of strong encapsulation enforcement. For a context to be valid, it must be in the set of currently visible contexts:

\[
\Gamma \vdash K : C(\overline{K}) \quad \text{method}(C(\overline{K}), m, T) = \langle \_\_\_X \rangle \\
K' \in \{K, \text{world, world.X}\} \cup \overline{X} \\
\Gamma; K; \langle m, T \rangle \vdash K' \\
\Gamma \vdash K : C(\overline{K}) \quad K' \in \overline{K} \cup \{K, \text{world, world.X}\} \\
\Gamma; K; \emptyset \vdash K'
\]

Only declared sub-contexts of this are valid:

\[
\Gamma \vdash K : C(\overline{K}) \quad X \in \text{subcontext}(C) \\
\Gamma; K; \bot \vdash K'
\]

Domination relationships are either stored in the environment, a produce of owner ordering, transitivity, world being the top context, or self domination:

\[
\begin{align*}
\Gamma \vdash K \leq K' & \in \Gamma & \Gamma \vdash K : C(\overline{K}) \\
\Gamma \vdash K \leq K' & \\quad \Gamma \vdash K \leq K_i \\
\Gamma \vdash K \leq \text{world} & \\quad \Gamma \vdash K \leq K \\
\Gamma \vdash K \leq K'' & \\quad \Gamma \vdash K'' \leq K' \\
\end{align*}
\]

To validate a type we ensure the number of actual context parameters matches the number of formal context parameters and that the supplied contexts are valid:

\[
\text{class } C[\overline{X}] \ldots \quad |\overline{X}| = |K| \\
\Gamma; K; \Delta \vdash C(\overline{K}) \\
\Gamma; K; \Delta \vdash C(\overline{K})
\]

We make sub-typing transitive:

\[
\Gamma \vdash T <: T' \quad \vdash T' <: T'' \\
\Gamma \vdash T <: T''
\]

We permit type coercion through sub-typing:

\[
\text{class } C_1[\overline{X}_1] \text{ extends } C_2 \ldots \\
\text{class } C_2[\overline{X}_2] \ldots \\
\Gamma \vdash C_2(\overline{K}_1, \overline{X}_2) <: D(\overline{K'}) \\
\Gamma \vdash C_1(\overline{K}) <: D(\overline{K'})
\]

7.5 Type System Safety

The proofs of progress and preservation for this system are completely standard as we do not seek to enforce properties on the heap other than the compile time enforced tree ordering of contexts. A formal presentation of the operational semantics of this language as well as a proof of progress and preservation is available in a separate technical report for interested readers [5].

8. PROOF OF THE SUFFICIENT CONDITIONS FOR LOOP PARALLELISM

Having completed the formal presentation of our type system, we return again to the sufficient conditions for parallelization we outlined in Section 3 for the stereotypical data parallel loop:

\[
\text{foreach}(\text{htmlspecialchars} \cdot \text{element in collection}) \{
\text{element.operation(arguments)};
\}
\]

Formally, we stated the collection must satisfy the comb property and the write set of the loop body may contain only this and the read set may contain this as well as any contexts $r$ such as $r$ and $c$ are unordered ($c < r$). In this section we aim to formally prove that this is sufficient to safely parallelize the loop without synchronization.

We start by proving that if an expression, directly or indirectly, writes a field of an object then the owning context of the object, or one of the contexts which dominates it will appear in the write set of the expression. We prove this by induction over the rules for computing write effects (Section 7). The base case is the calculation of the write effects for a direct assignment to a field where the resulting write set explicitly has the owning object's context placed in the expression's write set. The other type rules recursively ensure that if any writes occur as a side-effect of a component expression, the other context or one of its ancestors is contained in the write set. The raise function, by construction, either returns the given context or a context which dominates the given context. This proves the capture of write effects by the system; an identical argument can be made for the reading of a field and the read set. This ensures that the effects on a method can be relied on to encompass all of the contexts which may be directly or indirectly read or written.
A loop can be parallelized provided no data or control dependencies exist between iterations. We assume in this paper that no implicit control dependencies, such as exceptions, exist. Data dependencies take one of three forms as previously described: output dependencies, flow dependencies, and anti-dependencies.

Assume, by way of contradiction, that an output dependence exists between iterations. The collection must contain two separate elements \( e_1 \) and \( e_2 \) such that \( e_1 \) operation (arguments) writes to a field of some object \( x \) and \( e_2 \) operation (arguments) writes to that same field of \( x \).

The write set of operation (arguments) may contain only this, so we know that \( e_1 \) operation (arguments) can only write to objects that are either \( e_1 \) or strictly dominated by \( e_1 \). Similarly, \( e_2 \) operation (arguments) can only write object that are either \( e_2 \) or strictly dominated by \( e_2 \). Figure 3 shows this set of relationships.

![Figure 3: The relationships between \( e_1, e_2, \) and \( x \)](image)

We know from the comb property that \( e_1 \neq e_2 \), so \( x \) is not \( e_1 \) or \( e_2 \). The object \( x \) must, therefore, be strictly dominated by both \( e_1 \) and \( e_2 \). Each object is owned by only one object. If \( x \) is strictly dominated by \( e_1 \) and \( e_2 \), it must be the case that either \( e_1 \) dominates \( e_2 \) or \( e_2 \) dominates \( e_1 \). But, \( e_1 \) and \( e_2 \) are both directly owned by the same context \( c \), which provides the contradiction.

Assume now, by way of contradiction, that a flow dependence exists. The collection must contain two elements \( e_1 \) and \( e_2 \) such that \( e_1 \) operation (arguments) writes to some field \( x \) and \( e_2 \) operation (arguments) reads that same field \( x \). We know from the final step of the proof above that there is no \( x \) that is part of both \( e_1 \)’s and \( e_2 \)’s representation.

The only other source of such a flow dependence would be if \( e_2 \) operation (arguments) reads the same field \( x \) via some context \( r \) such that \( r \) is unordered with respect to \( e_1 \)’s and \( e_2 \)’s owning context \( c \). Figure 4 shows the relationship between \( c,e_1,e_2, \) and \( x \).

![Figure 4: Relationship of \( e_1,e_2,c,r \), and \( x \) and the separation of \( c \) and \( r \) for the proof of the absence of flow dependencies](image)

So, \( x \) is dominated by \( e_1 \), which is dominated by \( c \). But \( x \) must also be dominated by \( r \) which is not possible as \( c << r \). Therefore, no flow dependence can exist. A mirror argument can be made to provide the absence of anti-dependencies.

9. RELATED WORK

In previous work [10] we developed a similar technique, the sufficient conditions failed to allow reading of shared state and so did not require sub-contexts or a runtime representation.

While there is little work that uses ownership techniques to discover exploitable inherent parallelism, there are a number of works which have contributed to the techniques used in our system. Sub-contexts and method level context parameters have been proposed by others for similar purposes, but we introduce them in a simplified manner without regard for strict encapsulation enforcement to prevent complicating the context system more than is strictly necessary.

Ownership Domains is an Ownership Types derivative system created by Aldrich and Chambers [1] for which Smith has developed an effects system [12]. Ownership Domains is a very powerful and very expressive type system which was primarily designed to express encapsulation properties. It makes extensive use of sub-contexts and explicit access permissions. Ownership Domains also has sub-contexts similar to those proposed in our system. The major difference is that sub-contexts in Ownership Domains each have their own access policy and they can be visible outside of the class. This is a very powerful system and can express a very large number of encapsulation relationships. Our system was purposely designed not to provide such power; we use the sub-contexts within an object to disambiguate between different logical segments of data belonging to the same object. Our focus on reasoning about parallelism necessitates simplicity in the memory model being used; large numbers of complex features can significantly complicate reasoning and analysis and we deliberately avoid these features with an aim to simplifying the task of discovering inherent parallelism.

Smith’s effects system for Ownership Domains is very similar in syntax to that which we have proposed. The difference in the operation of our effects system is largely caused by the underlying ownership type systems; our limited use of sub-contexts and the more orthodox ownership and context structure employed by our system allow us to introduce our raise operation which allows for effect abstraction; a feature not found in Smith’s system. We feel our context abstraction facilities are one of the key techniques necessary to reduce the complexity of parallelism reasoning and avoid revealing implementation details through effects.

The use of context parameters on methods, owner polymorphic methods, is not a new idea; Clarke [9], Wrigstad [14], and others have used the same techniques and we gratefully acknowledge their contribution. One key point to note is that these other proposals have been criticized because the polymorphism prevents static strong encapsulation enforcement; we are not concerned by this because we do not seek to enforce strong encapsulation and the contexts are bound at the invocation sites, which is where we need to be concerned about the effects of the method.

Boyapati, Lee, and Rinard developed a system which used ownership types to reason about the absence of data races and deadlocks in explicitly parallel code; more specifically, ensuring the safe use of locking protocols [2]. Their deadlock avoidance system enforced that locks are acquired in ownership tree order starting at the world. To prevent race conditions their system tracked which contexts were accessed by each method and required synchronization when they over-
lapped. This system is very useful when verifying lock based concurrency control in already parallelized systems. Their system still requires the user to manually synchronize data access and so identify where exploitable parallelism exists. Their system requires locks on all state accesses as there is no distinction between reads and writes. Their system does not permit multiple concurrent reads for the same reason and does not possess the raise operation that we have developed to prevent implementation exposure.

JOE3, proposed by Østlund et al, is a very powerful language which provides for immutability and read-only references. They provide many of the same language features provided by our system including a more powerful form of sub-contexts which can be nested and have reduced permissions to access their parent context. The key distinctions between our system and JOE3 are related to system complexity. JOE3 provides very powerful language features, but requires programmers and software architects to very carefully consider what permissions each class and method needs to have with respect to different contexts. The restrictions placed at the class level can be revoked and replaced by method level permissions, but this makes deciding the permissions of different aspects of a class difficult. JOE3 does provide unique references and other features not found in current imperative object-oriented languages, but with the cost of additional complexity. We have chosen to keep our system as simple as possible to make it easy to understand and use. Further, our system can abstract effects, an important feature; we are not sure if the same could be done with the effects of JOE3 due to the different permissions observed at different levels etc.

Lastly, Lu Potter and Xue’s Oval is an ownership types based language which captures program invariants. Their system, like Spec#, can detect a number of different types and sources of bugs that cannot be detected by current imperative object-oriented languages. Unfortunately, while these systems are very powerful and can detect bugs which our system cannot, we feel that the use of invariants makes the system significantly harder to use. Getting programmers to encode all of the invariants which must hold is difficult and error prone. Further, there is no easy way to verify if the invariants provided are ‘correct’ whereas our system can ensure that the declared effects and the code effects are consistent and alert the user when they are not.

10. CONCLUSION & FUTURE WORK

In this paper we have presented a system based on ownership types for reasoning about the data dependencies in modern imperative object-oriented programs. Specifically, we have contributed an effects system compatible with modern component software development methods as well as sufficient conditions for the exploitation of data parallel loops and task parallelism.

We are currently in the process of implementing a parser and type checker to incorporate the ideas presented in this paper into an industrial strength language. This will enable us to obtain greater insight into the feasibility and ease of use of such a system. We will use these experiences to guide the extension of our system to handle additional common use cases and real-world application complexities. We also plan to investigate the degree to which we can automate or assist the programmer in applying ownership annotations to existing codes.

11. REFERENCES