Verifying Safety Policies with Size Properties and Alias Controls

Wei-Ngan Chin[1,2], Siau-Cheng Khoo[1], Shengchao Qin[3*], Corneliu Popeea[1], and Huu Hai Nguyen[2]
[1]Department of Computer Science, National University of Singapore
[2]Computer Science Programme, Singapore-MIT Alliance
[3]Department of Computer Science, University of Durham
{chinwn|khoosc|popeeaco|nguyenh2}@comp.nus.edu.sg shengchao.qin@durham.ac.uk

Abstract

Many software properties can be analysed through a relational size analysis on each function's inputs and outputs. Such relational analysis (through a form of dependent typing) has been successfully applied to declarative programs, and to restricted imperative programs; but it has been elusive for object-based programs. The main challenge is that objects may mutate *and they may be* aliased*. In this paper, we show how safety policies of programs can be analysed by tracking size properties of objects and be enforced by objects' invariants and the preconditions of methods. We propose several new ideas to allow both* mutability *and* sharing *of objects, whilst aiming for* precision *in our analysis. We introduce the concept of* size-immutability *to facilitate sharing, and also a set of* alias controls *to track unaliased objects whose size properties may change. We formalise our results through a set of advanced type checking rules for an object-based imperative language. We re-affirm the utility of the proposed type system by showing how a variety of software properties can be automatically verified according to size-inspired safety policies.*

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Dependent Types*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Languages, Theory, Verification

*This work was mostly done while the author was affiliated with the Singapore-MIT Alliance, National University of Singapore.

Keywords

Object-based programs, Safety verification, Dependent types, Size properties, Alias control

1. Introduction

Size properties are extremely useful in program analysis to support program optimization and verification. Each piece of data structure typically has one or more associated size properties. For simple types, such as `int`, `bool` or `file`, size property can capture their values (or abstract states). For aggregate types, size may refer to the length/height of linked structures, cardinality of collections or bounds of arrays. We may also ascribe size property to memory utilization in order to estimate space required for execution. For functions, size property can capture relations between input and output sizes, input preconditions, and also size invariance for parameters in recursion[10].

Past research into size properties has been most explored for functional [23, 22, 36, 10] and logic programming languages [34, 3]. Such properties have been applied to a range of important problems, including: (i) array bound check elimination [38, 39], (ii) memory space estimates for both region model [22] and heap-with-reuse model [21], (iii) complexity analysis [18], (iv) termination analysis[3, 26], and (v) support for safer programming where size invariants (e.g. balanced AVL trees) are guaranteed at compile time[36]. These applications are important towards supporting the goal of more reliable (and rigorous) software engineering processes. They constitute a realization of the automated verification for size-based software contracts.

To track size properties accurately, a *relational analysis* is required. This is expressible through a form of *dependent typing*[36], known as *sized type*[23, 10], that has been applied to declarative paradigms. To the best of our knowledge, no one has successfully designed a relational size analysis (or corresponding dependent type system) for static reasoning of object-based imperative programs.

Two key problems with objects are *mutability* and *aliasing*. Objects may mutate and their property changes be tracked. Objects may also alias with each other, and this must be analysed and/or be controlled. To highlight the problem of aliasing in the presence of mutation, consider the following `Int` object type expressed as an abstract data type (ADT), with an increment method.

```
adt Int { int val ;
void incr(Int x, int v) { x.val = x.val + v } }
```

Each object of the Int type has a primitive integer val field, whose size we may wish to track. To do so, we introduce an annotated object type, say $\mathtt{Int}\langle\mathtt{n}\rangle$, where n is a size variable for capturing the value of the val field via n=m and with true as the object's invariant, as follows:

$\mathtt{adt\ Int}\langle\mathtt{n}\rangle\ \mathtt{where\ n{=}m\ ;true}\ \{\ \mathtt{int}\langle\mathtt{m}\rangle\ \mathtt{val}\ ;\cdots\}$

Consider two variables a and b of the Int type. We can provide an annotated type for these variables using $\mathtt{a} :: \mathtt{Int}\langle\mathtt{x}\rangle, \mathtt{b} :: \mathtt{Int}\langle\mathtt{y}\rangle$, where x and y denote the sizes of current objects referenced by a and b, respectively. Consider now a program fragment with aliasing and updates.

```
Int a = new Int(5);    // x' = 5
Int b = a;             // y' = x ∧ x' = x
incr(a, 1);            // x' = x + 1
```

Ideally, we would like to perform size analysis, locally for each statement, as shown on the RHS. This would result in the following state $\mathtt{x}' = 6 \wedge \mathtt{y}' = 5$, after composing the effects of the three statements. However, this reasoning is unsound as we did not take alias relationships into account. Note the use of the prime notation (e.g. $\mathtt{x}', \mathtt{y}'$) to denote new states of the size variables after each statement.

Accurately tracking the sizes of mutable objects requires the aliasing problem to either be *carefully analysed* or be *tightly controlled*. In this paper, we chose the latter path, as analysing global aliases could quickly become unmanageable. Our main contributions are:

- We have successfully designed a **sized type system for objects**, which is able to track size information accurately. To the best of our knowledge, no one has applied such an advanced relational analysis to object-based imperative programs before.
- Each object declaration is expressed as an ADT. Our motivation for using ADTs is to allow each set of software properties to be specified via a **safety policy** (or **protocol**), whose correct usage can be automatically verified against the *size invariant* of each ADT and the *preconditions* on its library of operations.
- Our approach provides an elegant balance between *mutability* and *sharing*, while working towards accurate size tracking. We achieve this using a novel property, called **size-mutability**, obtained with a set of alias controls. We use *read-only* annotation for size-immutability, and *uniqueness* and *lent* annotations to limit aliasing for mutable size properties.
- We formalise a set of advanced type checking rules with alias controls and size constraints. We have implemented a **prototype** to validate our proposal and have also proven key **safety theorems** to confirm the desired properties of well-typed programs.

Sec 2 proposes a core imperative language with objects, and highlights how size properties are captured via an annotated type system. Sec 3 introduces the basic concept of size-mutability and how to track it using alias controls. Sec 4 shows how safety protocols may be specified through size invariants and methods' preconditions in ADTs. Sec 5 presents a set of advanced size-checking type rules, followed by key correctness properties. Sec 6 describes related work, followed by a short conclusion.

2. Kernel Language

We design a simple kernel language, called OIMP, with alias and size annotations. We present an object-based rather than an object-oriented language to focus on the key issues for the size analysis of objects. We intend for OIMP to be used as a backend to existing programming languages, such as Java, C or C++. Its syntax is given in Fig. 1. Note that the suffix notation y^* denotes a list of zero or more distinct syntactic terms that are suitably separated. For example, $(t\ v)^*$ denotes $t_1\ v_1, \ldots, t_n\ v_n$ where $n{\geq}0$.

$$
\begin{array}{rl}
P & ::= \mathit{prim}^*\ \mathit{user}^*\ \mathit{meth}^* \\
\mathit{prim} & ::= \mathtt{adt}\ \mathit{pn}\langle n_1,\ldots,n_p\rangle\ \mathtt{where}\ \{n^*\};\ \phi_I\ \{\mathit{pmeth}^*\} \\
\mathit{user} & ::= \mathtt{adt}\ \mathit{cn}\langle n_1,\ldots,n_p\rangle\ \mathtt{where}\ \phi\ ;\ \phi_I\ \{\mathit{field}^*\ \mathit{meth}^*\} \\
\mathit{pn} & ::= \mathtt{void}\ \mid\ \mathtt{bool}\ \mid\ \mathtt{int}\ \mid\ \mathtt{float}\ \mid\ \cdots \\
\mathit{pmeth} & ::= t\ \mathit{mn}\ ((t\ v)^*)\ \mathtt{where}\ \phi_{\mathit{pr}}\ ;\ \phi_{\mathit{po}} \\
\mathit{meth} & ::= t\ \mathit{mn}\ ((t\ v)^*)\ \mathtt{where}\ \phi_{\mathit{pr}}\ ;\ \phi_{\mathit{po}}\ \{e\} \\
\tau & ::= \mathit{cn}\ \mid\ \mathit{pn} \qquad t ::= \tau\langle n^*\rangle\mathtt{@}A \\
w & ::= v\ \mid\ v.f \qquad \mathit{field} ::= t\ f \qquad A ::= \mathtt{U}\ \mid\ \mathtt{L}\ \mid\ \mathtt{S}\ \mid\ \mathtt{R} \\
e & ::= (\mathit{cn})\ \mathtt{null}\ \mid\ k\ \mid\ w\ \mid\ w = e\ \mid\ \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \\
 & \mid\ \mathtt{new}\ \mathit{cn}(v^*)\ \mid\ \mathit{mn}\ (v^*)\ \mid\ t\ v = e_1\ ;\ e_2 \\
 & \mid\ \mathtt{bind}\ v\ \mathtt{of}\ (v_1,..,v_n)\ \mathtt{in}\ e \\
\phi \in & \mathbf{F} \qquad \textit{(Presburger Size Constraint)} \\
 & ::= b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists n \cdot \phi \mid \forall n \cdot \phi \\
b \in & \mathbf{BExp} \qquad \textit{(Boolean Expression)} \\
 & ::= \mathtt{true} \mid \mathtt{false} \mid \alpha_1{=}\alpha_2 \mid \alpha_1{<}\alpha_2 \mid \alpha_1{\leq}\alpha_2 \\
\alpha \in & \mathbf{AExp} \qquad \textit{(Arithmetic Expression)} \\
 & ::= c \mid n \mid c*\alpha \mid \alpha_1{+}\alpha_2 \mid -\alpha \mid \mathit{max}(\alpha_1,\alpha_2) \mid \mathit{min}(\alpha_1,\alpha_2) \\
 & \textit{where } c \in Z \textit{ is an integer constant}\ ;\ \ n \in SV \textit{ is a size variable} \\
 & f \in Fd \textit{ is a field name}\ ;\ \ v \in Var \textit{ is an object variable}
\end{array}
$$

Figure 1. Syntax for the OIMP language

Local variable declaration is supported by block structure of the form: $(t\ v = e_1; e_2)$ with e_2 denoting its result. For convenience, we may also use τ instead of t and rely on fresh size variables and alias defaults for annotations. We assume a call-by-value mechanism for OIMP, where values (primitives or references) are passed as arguments to parameters of methods. For simplicity, we do not allow parameter updates (or re-assignments). There is no loss of generality, as we can always copy parameters to local variables for updating, without altering the observable behaviour of method calls. Each user-defined method declaration *meth* contains a precondition ϕ_{pr} and a postcondition ϕ_{po}. The former must be satisfied for each caller, while the latter must be ensured by the method's body. For size constraints used (e.g ϕ_{pr} and ϕ_{po}), we restrict to Presburger arithmetic, as decidable (and practical) constraint solvers exist, e.g. [32]. For each reference (variable, parameter or field) of an object type, we add an alias control, named A, as follows: $\tau\langle\mathtt{n}_1,\ldots,\mathtt{n}_\mathtt{m}\rangle\mathtt{@A}$. Each alias control could either be R, U, S or L to denote *read-only*, *unique*, *shared* or *lent-once*, respectively. They are primarily used to track the mutability of size properties for objects and uniqueness of their references, and will be elaborated later in Sec 3.

The OIMP language has been kept simple to make easier the formulation of static and dynamic semantics. Typical language constructs, such as multi-declaration block, sequence, calls with complex arguments, *etc.* can be automatically translated to constructs in OIMP. Also, loops can be viewed as syntactic abbreviations for tail-recursive methods, and are supported by our analysis.

2.1 Primitive ADTs

We provide two kinds of object type, namely primitive (*prim*) type or user-definable (*user*) type. Both are expressed using the ADT mechanism to allow safety policies on objects to be specified. For each primitive ADT, we specify a set of size variables $n_1, \ldots, n_p$ with a size invariant ϕ_I that holds for each object, and $\{n^*\}$ to capture a subset of the size variables that are mutable. We also specify a set of primitive method declarations that are trusted to meet its postcondition ϕ_{po}, whenever its precondition ϕ_{pr} is satisfied. An example is the `bool` primitive ADT, with no mutable size variables and $0 \leq n \leq 1$ as its size invariant.

$$
\begin{array}{l}
\texttt{adt bool}\langle\texttt{n}\rangle\ \texttt{where}\ \{\}\ ;\ 0\leq\texttt{n}\leq 1 \\
\{\ \texttt{bool}\langle\texttt{r}\rangle\texttt{@S not}(\texttt{bool}\langle\texttt{m}\rangle\texttt{@S})\ \texttt{where} \\
\quad \texttt{true}; \texttt{m}'{=}\texttt{m}\wedge(\texttt{m}{=}0\wedge\texttt{r}{=}1\vee\texttt{m}{=}1\wedge\texttt{r}{=}0); \cdots\ \}
\end{array}
$$

Another example is the array ADT, with a safety policy that no array bound violations ever occur, as given below:

$$
\begin{array}{l}
\texttt{adt Array}\langle\texttt{s}\rangle\ \texttt{where}\ \{\};\ \texttt{s}{>}0 \\
\{\texttt{Array}\langle\texttt{s}\rangle\texttt{@U newArray}(\texttt{int}\langle\texttt{n}\rangle\texttt{@S a}, \texttt{int}\langle\texttt{v}\rangle\texttt{@S val}) \\
\quad \texttt{where n}{>}0\ ;\ \texttt{s}{=}\texttt{n}\wedge no\mathcal{X}\{\texttt{n},\texttt{v}\}; \\
\ \texttt{int}\langle\texttt{r}\rangle\texttt{@S sub}(\texttt{Array}\langle\texttt{s}\rangle\texttt{@S a}, \texttt{int}\langle\texttt{i}\rangle\texttt{@S idx}) \\
\quad \texttt{where}\ 0 \leq \texttt{i} < \texttt{s}\ ;\ no\mathcal{X}\{\texttt{s},\texttt{i}\}\ ; \\
\ \texttt{void}\langle\rangle\texttt{@S update}(\texttt{Array}\langle\texttt{s}\rangle\texttt{@S a}, \texttt{int}\langle\texttt{i}\rangle\texttt{@S idx}, \\
\quad \texttt{int}\langle\texttt{v}\rangle\texttt{@S val})\ \texttt{where}\ 0 \leq \texttt{i} < \texttt{s}\ ;\ no\mathcal{X}\{\texttt{s},\texttt{i},\texttt{v}\}\ ; \\
\ \texttt{int}\langle\texttt{r}\rangle\texttt{@S len}(\texttt{Array}\langle\texttt{s}\rangle\texttt{@S a})\ \texttt{where true}\ ;\ \texttt{r}{=}\texttt{s}\wedge\texttt{s}'{=}\texttt{s};\ \}
\end{array}
$$

Safe array bound is guaranteed by the precondition $0\leq \texttt{i} < \texttt{s}$ for the `sub` method (to retrieve an element of the array) and `update` method (to modify a location of the array), while $\texttt{n} > 0$ is for the `newArray` method (to create a new array of positive size). Furthermore, the array's size is never changed by any of the operations. This may be specified through $no\mathcal{X}$ which returns a formula where the original and prime variables are made equal, e.g. $no\mathcal{X}\{\texttt{s},\texttt{i}\}\equiv(\texttt{s}'{=}\texttt{s}\wedge\texttt{i}'{=}\texttt{i})$. As every call's precondition is checked, no array bound violation occurs. The binary search method below adheres to this safe array policy, with the help of the precondition $0\leq\texttt{i}\wedge\texttt{j}<\texttt{s}$.

$$
\begin{array}{l}
\texttt{int}\langle\texttt{r}\rangle\texttt{@S bs}(\texttt{Array}\langle\texttt{s}\rangle\texttt{@S a}, \texttt{int}\langle\texttt{i}\rangle\texttt{@S i}, \texttt{int}\langle\texttt{j}\rangle\texttt{@S j}, \texttt{int}\langle\texttt{v}\rangle\texttt{@S} \\
\quad \texttt{val})\ \texttt{where}\ 0\leq\texttt{i}\wedge\texttt{j}<\texttt{s}\ ;\ no\mathcal{X}\{\texttt{s},\texttt{i},\texttt{j},\texttt{v}\}\wedge(\texttt{r}{=}{-}1\vee\texttt{i}\leq\texttt{r}\leq\texttt{j})\ \{ \\
\ \texttt{if j}{<}\texttt{i then}\ {-1}\ \texttt{else}\ \{\ \texttt{int m}{=}(\texttt{i}{+}\texttt{j})/2;\ \texttt{int p}{=}\texttt{sub}(\texttt{a},\texttt{m}); \\
\qquad \texttt{if p}{<}\texttt{val then bs}(\texttt{a},\texttt{i},\texttt{m}{-}1,\texttt{val}) \\
\qquad \texttt{else if p}{>}\texttt{val then bs}(\texttt{a},\texttt{m}{+}1,\texttt{j},\texttt{val})\ \texttt{else m}\ \}\ \}
\end{array}
$$

For some problems, e.g. sparse matrix, the indexes may come from another array, called an indirection array. To handle such programs, we can provide:

$$
\begin{array}{l}
\texttt{adt ArrayI}\langle\texttt{s},\texttt{mn},\texttt{mx}\rangle\ \texttt{where}\ \{\texttt{mn},\texttt{mx}\};\ \texttt{s}{>}0\wedge\texttt{mn}\leq\texttt{mx}\ \{ \\
\texttt{void}\langle\rangle\texttt{@S update}(\texttt{ArrayI}\langle\texttt{s},\texttt{mn},\texttt{mx}\rangle\texttt{@L a}, \\
\quad \texttt{int}\langle\texttt{i}\rangle\texttt{@S idx}, \texttt{int}\langle\texttt{v}\rangle\texttt{@S val})\ \texttt{where}\ 0 \leq \texttt{i} < \texttt{s}\ ; \\
\qquad \texttt{mx}'{=}\texttt{max}(\texttt{mx},\texttt{v})\wedge\texttt{mn}'{=}\texttt{min}(\texttt{mn},\texttt{v})\wedge no\mathcal{X}\{\texttt{s},\texttt{i},\texttt{v}\} \\
\texttt{int}\langle\texttt{r}\rangle\texttt{@S sub}(\texttt{ArrayI}\langle\texttt{s},\texttt{mn},\texttt{mx}\rangle\texttt{@L a}, \texttt{int}\langle\texttt{i}\rangle\texttt{@S idx}) \\
\quad \texttt{where}\ 0 \leq \texttt{i} < \texttt{s}\ ;\ \texttt{mn} \leq \texttt{r} \leq \texttt{mx}\wedge no\mathcal{X}\{\texttt{s},\texttt{mn},\texttt{mx},\texttt{i}\}; \cdots\ \}
\end{array}
$$

Apart from `s` which captures the size of the array, we also have `mn` and `mx` to capture the min/max values for the elements in the array. The last two size variables are *mutable* and may be changed by the `update` method, but is guaranteed to satisfy the size invariant $\texttt{mn}\leq\texttt{mx}$. Furthermore, we can bound the result of `sub` method whenever indexes are retrieved from an indirection array.

2.2 User-Defined ADTs

Each user-defined ADT is specified with a set of fields that are visible only to its accompanying methods. (In particular, each `new` or `bind` construct may only be used within its ADT declaration.) For each ADT declaration, we specify a set of size variables $n_1, \ldots, n_p$ that can be tracked via the constraint ϕ, that is limited to $\bigwedge_{i=1}^{p} n_i{=}\alpha_i$ whereby $V(\alpha_i)\cap\{n_1, \ldots, n_p\}{=}\emptyset$. This defines size properties that depend solely on object components. Another constraint ϕ_I is used to capture the size invariant of each instance of this object type. For example, a user-defined `Bool` object type can be specified in terms of a primitive `bool` field, as follows. Note that this primitive field may be updated.

$$
\begin{array}{l}
\texttt{adt Bool}\langle\texttt{n}\rangle\ \texttt{where n}{=}\texttt{m}\ ;\ 0\leq\texttt{n}\leq 1\ \{\ \texttt{bool}\langle\texttt{m}\rangle\texttt{@S val}; \\
\ \texttt{void}\langle\rangle\texttt{@S Not}(\texttt{Bool}\langle\texttt{m}\rangle\texttt{@L v})\ \texttt{where true}; \\
\qquad \texttt{m}{=}0\wedge\texttt{m}'{=}1\vee\texttt{m}{=}1\wedge\texttt{m}'{=}0\ \{\texttt{b.val}{=}\texttt{not}(\texttt{b.val})\}; \cdots\ \}
\end{array}
$$

A more complex example is the binary tree with a size variable `s` to track the number of nodes, defined as:

$$
\begin{array}{l}
\texttt{adt Tree}\langle\texttt{s}\rangle\ \texttt{where s}{=}1{+}\texttt{s1}{+}\texttt{s2}\ ;\ \texttt{s}\geq 0 \\
\{\ \texttt{int}\langle\texttt{v}\rangle\texttt{@S val}\ ;\ \texttt{Tree}\langle\texttt{s1}\rangle\texttt{@U left}\ ;\ \texttt{Tree}\langle\texttt{s2}\rangle\texttt{@U right}; \cdots\ \}
\end{array}
$$

Alternatively, if we desire AVL trees, we could add a size variable `h` for height and another invariant $-1\leq\texttt{h1}-\texttt{h2}\leq 1$ to enforce a balanced height requirement, as shown below. This invariant can be viewed as a safety policy for our type system to ensure that only proper AVL trees (with this invariant) are constructed.

$$
\begin{array}{l}
\texttt{adt AVLTree}\langle\texttt{h},\texttt{s}\rangle\ \texttt{where h}{=}1{+}\texttt{max}(\texttt{h1},\texttt{h2})\wedge\texttt{s}{=}1{+}\texttt{s1}{+}\texttt{s2} \\
\quad ;\ \texttt{h}\geq 0\wedge\texttt{s}\geq 0\wedge({-1}\leq\texttt{h1}{-}\texttt{h2}\leq 1) \\
\{\ \texttt{int}\langle\texttt{v}\rangle\texttt{@S val}\ ;\ \texttt{AVLTree}\langle\texttt{h1},\texttt{s1}\rangle\texttt{@U left}\ ; \\
\ \texttt{AVLTree}\langle\texttt{h2},\texttt{s2}\rangle\texttt{@U right}; \cdots\ \}
\end{array}
$$

Size invariant is safely verified within each ADT, while a method's precondition is verified for each of its callers.

2.3 A Special Construct

Our kernel language includes a special construct, called **bind**, which is used to associate the fields $f_1, \ldots, f_n$ of an object v to a set of local variables $v_1, \ldots, v_n$. This binding is valid over the scope of an expression e and results in local variables being used as synonyms to the corresponding fields. Unlike the usual pattern-matching construct (see [37]), **bind** is based on a pass-by-name (instead of pass-by-value) mechanism. Furthermore, it is used *only* for the purpose of analysis and do not cause runtime overheads. This construct is particularly crucial for tracking the size properties of objects that depend on multiple components. For example, consider a method to insert an element into a binary search tree.

```
void⟨⟩@S insert(Tree⟨s⟩@L t, int⟨c⟩@S i)
    where s>0 ; c′=c
  { if i<t.val then if t.left=null then
                         t.left = new Tree(i, null, null)
                       else insert(t.left, i)
    else if t.right=null then
                         t.right = new Tree(i, null, null)
         else insert(t.right, i) }
```

With the field access construct, we can only capture the size relation between an object and *a single field*. With the **bind** construct, we can capture the size relation between an object and *all its fields*. Hence, we can capture a more precise size mutation for size variables that depend on multiple fields, as illustrated in the following alternative definition of `insert`. The local variables `w, l, r` are associated with the corresponding fields, in the same order as their declaration.

```
void⟨⟩@S insert(Tree⟨s⟩@L t, int⟨c⟩@S i)
    where s>0 ; c′=c∧s′=s+1
  { bind t of (w, l, r) in
    if i<w then if l=null then l = new Tree(i, null, null)
             else insert(l, i)
    else if r=null then r = new Tree(i, null, null)
         else insert(r, i) }
```

Despite being special, we can automatically insert **bind** constructs into our program as follows. For each field access $v.f$ of a variable v, we identify the largest expression scope that anticipates a field access but does not access v itself (unless it is in shared mode), before introducing a binding for v and its fields $v.f_1, \ldots, v.f_n$ through a fresh set of local variables.

3. Alias Controls for Size Tracking

We adopt four alias controls from [7, 1], namely read-only (`R`), lent (`L`), shared (`S`) and unique (`U`). Each reference in a location (field, parameter or local variable) can be marked with any of the annotations, except that `L`-mode applies only to parameters. While the alias controls are adapted from [1, 7], there are differences in how the type rules are being checked, as described later in Sec 5. Formal definitions of these alias properties are given below.

DEFINITION 1 (READ-ONLY). *A location that is marked as* **read-only** *(denoted by* `R`*) can be initialized with a reference, but cannot be changed thereafter via the location.*

DEFINITION 2 (UNIQUENESS). *A reference to an object is* **unique** *(denoted by* `U`*) if it is the* **only** *reference to the object at the current scope.*

DEFINITION 3 (SHARED). *A reference (from a location) to an object is* **shared** *(denoted by* `S`*) if it may be globally aliased with other reference(s).*

In classical alias annotation work[1], the `L`-mode is used to describe variables whose references do not escape their declaration blocks. Such locations are candidates for lending, as the uniqueness of source objects are not compromised. As a result, each object can be lent out multiple times. The uniqueness property is always restored once all locations that are lent out (including transitive lending) are dead. However, the policy of allowing many lendings for a single object, and of allowing lents to be reassigned to other lent locations, may cause aliasing problems to re-surface. To avoid this problem, we propose a refined policy, called *lent-once*[1]:

DEFINITION 4 (LENT-ONCE). *A parameter that is marked as* **lent-once** *(denoted by* `L`*) has the following properties: (i) its value does not escape the method, (ii) the lent-once parameter is never re-assigned, (iii) it has exclusive access to the object in its method's scope.*

The last two conditions prevent aliases from occurring in the scope of each lent-once location. As we exclude local variable from the `L`-mode, call sites are the *only* places where lendings can occur. While this lent-once policy may appear restrictive, its shortcoming can be addressed by *preprocessing* (converting lent to lent-once, where possible) and *intersection types*[30] (allowing different parameter aliasing for each call site).

3.1 Alias Subtyping

As we deal with the types of expressions, our subtype relation (shown below) need not include `R`. Each value retrieved from an `R`-field is automatically casted to the `S`-mode. Note that `U`-mode references act as universal sources, while `L`-mode locations serve as sinks only. This subtyping effectively *prevents* an `S`-reference from being copied into a `U`-location. Unlike [1], we disallow $\mathtt{S} \leq_a \mathtt{L}$ to enforce non-aliasing of the `L`-mode parameters.

$$A \leq_a A \qquad \mathtt{U} \leq_a \mathtt{L} \qquad \mathtt{U} \leq_a \mathtt{S}$$

3.2 Size Mutability

Our main idea for tracking size properties is the concept of *size-mutability*. To help identify size properties that are immutable for objects, we shall use the *read-only* mode to protect certain fields from being changed. We classify each *field* and *size variable*, as follows:

DEFINITION 5 (SIZE-MUTABILITY FOR FIELD). *A field is classified as* size-immutable*, if it is marked with the* `R`*-mode. Otherwise, it is classified as* size-mutable.

DEFINITION 6 (SIZE-MUTABILITY FOR SIZE VARIABLE). *A* size variable *of an object type is* size-immutable *if it depends on only size-immutable variable(s) that come from size-immutable field(s). Otherwise, it is* size-mutable.

For some primitive type, like `int⟨n⟩`, its size variable is automatically classified as *size-immutable*, as its values cannot be changed after they have been created. For user-defined object type, like `Int⟨n⟩`, its field (size property) may change after its objects have been created and is therefore *size-mutable*, in general. Nevertheless, we can obtain more size-immutable fields/variables with the help of the `R`-mode. Consider:

```
adt Pair⟨a, b, c⟩ where a=p ∧ b=q ∧ c=p+q ; true
  { int⟨p⟩@S x; int⟨q⟩@R y; ··· }
adt List⟨n⟩ where n=m+1 ; n≥0
  { Int⟨v⟩@S val; List⟨m⟩@R next; ··· }
```

[1] Subsequent to our formulation, we found out that a similar policy, called *limited unique*, has already been proposed in [9], though their focus has been on the specification of such properties and not on a formal mechanism, such as the type-based one proposed here.

In `Pair`, the first field is size-mutable, while the second field is size-immutable. Of its three size variables $\{\mathtt{a},\mathtt{b},\mathtt{c}\}$, only `b` is size-immutable. This is because `b` depends on only size-immutable `q` which came from a read-only field. In `List`, we use `n` to track the length of the linked list, and this is made size-immutable by protecting the `next` field with the `R`-mode.

Objects may be freely shared without losing track of the properties of their size-immutable variables. However, the same cannot be said for size-mutable variables, as global aliasing can quickly invalidate local analysis. Our solution for objects with size-mutable fields to track, is to make their references *unique*. Uniqueness can guarantee that the references of these fields are never aliased.

For example, we can design another list structure whose `next` field can be updated but must be unique, as shown below. Note that uniqueness allows the length of this linked list to be tracked, even with mutation on its tail.

```
adt List2⟨n⟩ where n=m+1 ; n≥0
  { Int⟨v⟩@S val; List2⟨m⟩@U next; ··· }
```

3.3 Classification of Size Variables

Thus far, we have classified size variables into two main groups, namely size-mutable and size-immutable. However, the size-mutable group can be further classified into either *trackable* or *non-trackable* size variables, due to the *absence* or *presence* of global aliasing, respectively. This classification of size variables can be inferred from the objects' type declarations. We provide a pair of mutual recursive definitions, named $\mathbf{V}_{obj}$ and $\mathbf{V}_{field}$, to classify the size variables of annotated *object type* and *fields/parameters* (with suffixes I, T, and N to denote *immutable, trackable and non-trackable* respectively), as follows:

$$\frac{\mathbf{V}_{obj}(\tau\langle s^*\rangle) = (d_I, d_T, d_N)}{\mathbf{V}_{field}(\tau\langle s^*\rangle@\mathtt{R}) =_{df} (d_I, \emptyset, d_T \cup d_N)}$$

$$\frac{\mathbf{V}_{obj}(\tau\langle s^*\rangle) = (d_I, d_T, d_N)}{\mathbf{V}_{field}(\tau\langle s^*\rangle@\mathtt{S}) =_{df} (\emptyset, d_I, d_T \cup d_N)}$$

$$\frac{A \in \{\mathtt{U},\mathtt{L}\} \qquad \mathbf{V}_{obj}(\tau\langle s^*\rangle) = (d_I, d_T, d_N)}{\mathbf{V}_{field}(\tau\langle s^*\rangle@A) =_{df} (\emptyset, d_I \cup d_T, d_N)}$$

$$\frac{\mathtt{adt}\ pn\langle n^*\rangle\ \mathtt{where}\ MVar;\ \phi_I\ \{pmeth^*\} \in P}{\mathbf{V}_{obj}(pn\langle n^*\rangle) =_{df} (n^* - MVar, MVar, \emptyset)}$$

$$\frac{\begin{array}{c}\mathtt{adt}\ cn\langle n^*\rangle\ \mathtt{where}\ \phi\ ;\ \phi_I\ \{(t_i\ f_i)_{i:1..p}\ meth^*\} \in P \\ \mathbf{V}_{field}(t_i) = (d_i^I, d_i^T, d_i^N), \quad i \in \{1..m\} \\ n_I = depends(n^*, \bigcup_{i=1}^m d_i^I, \phi) \\ n_T = depends(n^*, \bigcup_{i=1}^m d_i^T, \phi) \\ n_N = depends(n^*, \bigcup_{i=1}^m d_i^N, \phi)\end{array}}{\mathbf{V}_{obj}(cn\langle n^*\rangle) =_{df} (n_I - (n_T \cup n_N), n_T - n_N, n_N)}$$

For each field with size variables s^*, we extract from n^* those that depend on some s^* via ϕ. This dependency is defined below. Each primitive constraint from ϕ is of the restricted form $n = \alpha$. We define V to return all free size variables in a formula. For example, $V(\mathtt{x}'{=}\mathtt{z}+1 \wedge \mathtt{y}{=}2) = \{\mathtt{x}', \mathtt{y}, \mathtt{z}\}$. We extend it to annotated type and type environment, e.g. $V(\tau\langle n^*\rangle@A) = \{n^*\}$.

$$depends(n^*, s^*, \phi) =_{df} \bigcup_{(n=\alpha)\in\phi} \{n \mid (V(\alpha)\cap s^*) \neq \emptyset\}$$

Given $\phi_P = (\mathtt{a}{=}\mathtt{p} \wedge \mathtt{b}{=}\mathtt{q} \wedge \mathtt{c}{=}\mathtt{p}{+}\mathtt{q})$ from the `Pair` object definition, we can infer that $\{\mathtt{b}\}$ is size-immutable, $\{\mathtt{a},\mathtt{c}\}$ are trackable size-mutable and $\emptyset$ denotes an empty set of non-trackable size-mutable variables.

$$\begin{array}{ll} n_I = depends([\mathtt{a},\mathtt{b},\mathtt{c}],[\mathtt{q}],\phi_P) & = \{\mathtt{b},\mathtt{c}\} \\ n_T = depends([\mathtt{a},\mathtt{b},\mathtt{c}],[\mathtt{p}],\phi_P) & = \{\mathtt{a},\mathtt{c}\} \\ \mathbf{V}_{obj}(\mathtt{Pair}\langle\mathtt{a},\mathtt{b},\mathtt{c}\rangle) & = (\{\mathtt{b}\},\{\mathtt{a},\mathtt{c}\},\emptyset) \end{array}$$

For each object type declaration that is recursive, we may derive definitions for $\mathbf{V}_{obj}$ and $\mathbf{V}_{field}$ that are circular, but they can still be analysed via fixed-point computation.

4. Specification of Safety Protocols

With the help of size (and alias) annotations, we can specify suitable safety policies for software through the use of *size invariants* for each object type and the use of *preconditions* for each method declaration. Apart from enforcing data structure invariant (e.g. AVL tree) and avoiding bound violations (e.g. safe array), we may also check if operations of an ADT are invoked according to a temporal protocol.

Let us highlight how we can specify the safety policy for a file protocol (defined below) that allows read/write from a file only after it has been opened. The state of each file is encoded by $\mathtt{s}{\in}1..3$ with $1, 2, 3$ to denote *initial*, *open*, and *close* states, respectively. (While the states of protocol are encoded using integer values, we stress that it is easy to use syntactic sugar to make protocol specification more natural to programmers. Furthermore, our use of integer domain to encode values of finite states is meant to support a more convenient relational analysis via Presburger constraints.) Once closed, no more read/write operations can be performed. These temporal requirements are enforced by the sized preconditions of each method. In general, not all the safety preconditions can be enforced directly, but we allow programmer to insert runtime tests to ensure conformance. For example, opening a file may fail if it does not exist and this situation can be co-related with the returned flag (of `open`) that can be tested at runtime.

```
adt File⟨s⟩ where {s}; 1≤s≤3 {
  File⟨s⟩@U newFile() where true; s=1
  bool⟨b⟩@S open(File⟨s⟩@L f, string⟨⟩@S name)
       where s=1; (b=1∧s′=2)∨(b=0∧s′=1)
  string⟨⟩@S read(File⟨s⟩@L f) where s=2; s′=2
  void⟨⟩@S write(File⟨s⟩@L f, string⟨⟩@S buf)
      where s=2; s′=2
  void⟨⟩@S close(File⟨s⟩@L f) where s=2; s′=3
  int⟨r⟩@S getState(File⟨s⟩@L f) where true; r=s∧s=s′ }
```

Capturing such a protocol as a primitive ADT assumes that its implementation has been separately verified against its stated interface. We can also rely on user-defined ADTs for the specification of safety protocols. An example is the following bounded buffer ADT which is guaranteed to *neither underflow nor overflow* each buffer of a given capacity.

```
adt Buffer⟨s, c⟩ where s=n∧c=a;
      0≤s≤c∧0≤h<c∧0≤t<c∧c = e {
  Array⟨a⟩@R arr; int⟨e⟩@R cap; int⟨h⟩@S h; int⟨t⟩@S t;
  Buffer⟨s, c⟩@U newBuffer(int⟨n⟩@S n)
      where n>0; s=0∧c=n∧n′=n
    { new Buffer(newArray(n, 0), n, 0, n−1) }
  void⟨⟩@S add(Buffer⟨s, c⟩@L b, int⟨n⟩@S n)
      where s<c; s′=s+1∧noX{c, n}
    { bind b of (a, c, h, t) in {
                update(a, h, n); h=mod(h+1, c)} }
  int⟨n⟩@S get(Buffer⟨s, c⟩@L b)
      where s>0; s′=s−1∧c′=c
    { bind b of (a, c, h, t) in {
                t=mod(t+1, c); sub(a, t)} } ··· }
```

Note the use of a cyclic array to store the contents of the buffer, with the help of the following `mod` primitive.

```
int⟨r⟩@S mod(int⟨a⟩@S x, int⟨b⟩@S y)
    where b>0 ; 0≤r<b∧noX{a, b}
```

This example illustrates how we can safely build a more sophisticated ADT from simpler ones. As an example of the use of this buffer ADT, we have the following method which can safely add `n` elements, provided that the precondition $\mathtt{s}+\mathtt{n}\leq\mathtt{c}$ is satisfied.

```
void⟨⟩@S addMany(Buffer⟨s, c⟩@L b, int⟨n⟩@S n,
    int⟨v⟩@S val) where s+n≤c
        ; s′=s+max(0, n)∧noX{c, n, v}
  { if n≤0 then ()
    else add(b, val); addMany(b, n−1, val) }
```

5. Verification via Sized Typing

We present type judgements for *expressions*, *method declarations*, *object declarations* and *programs* to check for their well-typedness, using relations of the form:

$$\Gamma; C; \Delta; \Theta \vdash e :: t, \Delta_1, \Theta_1 \qquad C \vdash_{meth} meth \qquad \vdash_{pmeth} pmeth$$
$$\vdash_{prim} prim \qquad \vdash_{user} user \qquad \vdash_P prim^* \; user^* \; meth^*$$

Note that Γ is a type environment mapping program variables to their annotated types; $\Delta(\Delta_1)$ denotes the size constraint that holds for the size variables associated with Γ (Γ and t) for expression e before (after) its evaluation; t is an annotated type. C captures a set of user-defined ADTs whose fields are visible. Also, $\Theta(\Theta_1)$ is used to hold the variables whose uniqueness have been consumed before (after) the evaluation of e. We shall refer to such references as *dead*, as they are not to be accessed. Take note that both $\Delta(\Delta_1)$ and $\Theta(\Theta_1)$ are tracked flow-sensitively.

The major syntax-directed type rules for these judgements are given in Fig 2, whereas the rest of the rules are given in our technical report [11]. Compared to [1], we have attained two key innovations. Firstly, we do not require a separate *last-use analysis* to determine when a unique reference (from a variable) becomes dead, as we keep track of consumed uniqueness via the dead set (denoted by Θ). Secondly, we do not require *destructive read* for field access (which nullifies the field after each read). Instead, we track the liveness of each unique field with the help of the `bind` construct and the dead set. These changes improve both the accuracy and usability of our alias analysis.

In the rest of this section, we highlight important aspects of our sized type system via examples. Before that, we introduce some preliminary definitions. The function *prime* takes a set of size variables and returns their primed version, e.g. $prime(\{s_1, \ldots, s_n\}) = \{s'_1, \ldots, s'_n\}$. Note that prime operation is idempotent, namely $v'' = v'$. We extend this to annotated type, type environment and even substitution. For example, we have $prime(\tau\langle n_1, \ldots, n_k\rangle) = \tau\langle n'_1, \ldots, n'_k\rangle$, and $prime\ [x \mapsto a, y \mapsto b] = [x' \mapsto a', y' \mapsto b']$. We use $n^* = fresh()$ to generate new size variables n^*. We extend it to annotated type, so that $\hat{t} = fresh(t)$ will return a new type $\hat{t}$ with the same underlying type as t but with fresh size variables instead. The function $equate(t_1, t_2)$ generates equality constraints for the corresponding size variables of its two arguments, assuming that they share the same underlying type. For example, $equate(\mathtt{Int}\langle r\rangle, \mathtt{Int}\langle s'\rangle){=}(r{=}s')$. The function $rename(t_1, t_2)$ returns a mapping instead, e.g. $rename(\mathtt{Int}\langle r\rangle, \mathtt{Int}\langle s'\rangle){=}(r{\mapsto}s')$. To extract the alias of an annotated type, we use $ann(\tau\langle v^*\rangle\mathtt{@}A) = A$. The function $inv(t)$ returns the size invariant of t (after quantifying the size variables of its fields), while $inv(\Gamma)$ extends to type environment. The function $unify(t_1, t_2)$ returns a new fresh type whose alias annotation is an upper bound of t_1 and t_2. Also, conditional is expressed as $\xi_1 \lhd b \rhd \xi_2 \;=_{df}\; if\, b\;\; then\;\; \xi_1\;\; else\;\; \xi_2$.

Updates of size constraint due to change of program state are effected by a sequential composition operator, $\circ_X$, with X denoting the set of size variables that are being updated. For example, if $\Delta{=}(a'{=}a{+}b \wedge b'{=}3 \wedge i{=}a')$, then we can obtain the following where the a size variable is being updated.

$$\begin{aligned}(\Delta\circ_{\{a\}} a'{=}i{+}1) &= \exists a_0 \cdot a_0{=}a{+}b{\wedge}b'{=}3{\wedge}i{=}a_0{\wedge}a'{=}i{+}1\\ &= b'{=}3{\wedge}i{=}a{+}b{\wedge}a'{=}i+1\end{aligned}$$

More formally, sequential composition is defined as:

$$\phi_1 \circ_X \phi_2 \;=_{df}\; \exists R \cdot \rho_1(\phi_1) \wedge \rho_2(\phi_2)$$

where $X = \{s_1, \ldots, s_n\}$ are size variables being updated
$R = \{r_1, \ldots, r_n\}$ are fresh size variables
$\rho_1 = \{s'_i \mapsto r_i\}_{i=1}^n \qquad \rho_2 = \{s_i \mapsto r_i\}_{i=1}^n$

5.1 Variable Read

The [VAR] rule is used to return an object via a reference captured by variable v. Here, a unique reference loses its uniqueness and must not be referenced again (unless it has been re-assigned). It is thus added to a *dead set* using $\Theta_1{=}\Theta{\cup}(\{v\}{\lhd}A{=}\mathtt{U}{\rhd}\emptyset)$ where A is the alias annotation for the type of v. This rule also returns a freshly annotated type $t_1{=}fresh(t)$ and a size constraint $\phi{=}equate(t_1, prime(t))$ to relate the resulting annotated type with that of variable v. Assuming $\Gamma{=}\{\mathtt{v}{::}\mathtt{Int}\langle\mathtt{d}\rangle\mathtt{@U}\}$, $\Delta{=}no\mathcal{X}\{\mathtt{d}\}$, we have:

$$\frac{\phi = (\mathtt{r}_1{=}\mathtt{d}') \qquad \mathtt{v} \notin \Theta \qquad \mathtt{U} \neq \mathtt{L}}{\Gamma; C; \Delta; \Theta \vdash \mathtt{v} :: \mathtt{Int}\langle\mathtt{r}_1\rangle\mathtt{@U}, \Delta{\wedge}\phi,\ \Theta \cup \{\mathtt{v}\}}$$

Note how the dead set has changed, and how the size property of the result (namely $\mathtt{r}_1$) is associated with the latest size variables (namely $\mathtt{d}'$) from the type environment. L-mode references cannot escape via this rule.

$$[\underline{\textsc{aux}}]\quad \frac{\begin{array}{c}\Gamma(v) = t \quad t_1 = \mathit{fresh}(t)\\ \phi = \mathit{equate}(t_1, \mathit{prime}(t))\end{array}}{\Gamma \vdash v{::}t_1, \phi, V(t)}$$

$$[\underline{\textsc{assign}}]\quad \frac{\begin{array}{c}\Gamma; C; \Delta; \Theta \vdash e :: t_1, \Delta_1, \Theta_1 \quad \Gamma \vdash v{::}t, \phi, Y\\ \mathit{ann}(t) \notin \{\mathtt{L}, \mathtt{R}\} \quad \vdash t_1 <: t, \rho \quad X = V(t_1) \cup V(t)\\ \neg\, \mathit{isParam}(v) \quad \Delta_2 = \exists X \cdot \Delta_1 \circ_Y \rho(\phi)\end{array}}{\Gamma; C; \Delta; \Theta \vdash v = e :: \mathtt{void}\langle\rangle\mathtt{@S}, \Delta_2, \Theta_1/v}$$

$$[\underline{\textsc{var}}]\quad \frac{\begin{array}{c}\Gamma \vdash v{::}t, \phi, Y \quad A{=}\mathit{ann}(t) \quad A{\neq}\mathtt{L}\\ v{\notin}\Theta \quad \Theta_1{=}\Theta{\cup}(\{v\}{\lhd}A{=}\mathtt{U}{\rhd}\emptyset)\end{array}}{\Gamma; C; \Delta; \Theta \vdash v :: t, \Delta \wedge \phi, \Theta_1}$$

$$[\underline{\textsc{bind}}]\quad \frac{\begin{array}{c}\mathtt{adt}\ cn\langle s^*\rangle\ \mathtt{where}\ \phi; \phi_I\ \{(t_i\ f_i)_{i:1..p}\ \ \mathit{meth}^*\} \in P\\ v{\notin}\Theta \quad \Gamma(v) = cn\langle n^*\rangle\mathtt{@}A \quad \rho_1 = ([\mathtt{U} \mapsto \mathtt{S}]{\lhd}A{\in}\{\mathtt{S}, \mathtt{R}\}{\rhd}[])\\ cn{\in}C \quad \rho_2{=}[s{\mapsto}n']^* \quad \rho_3{=}\rho_2{\uplus} \biguplus_{i=1}^{p} \mathit{rename}(t_i, \mathit{prime}(t_i))\\ \Gamma_1{=}\{v_i{::}\rho_1 t_i\}_{i=1}^{p} \quad \Gamma{-}\{v|A{\in}\{\mathtt{L}, \mathtt{U}\}\}{\cup}\Gamma_1; C; \Delta_1; \Theta \vdash e :: t, \Delta_2, \Theta_1\\ \Delta_1{=}\Delta{\wedge}(\rho_2\ \phi{\wedge}\phi_I){\wedge}\mathit{no}\mathcal{X}(\Gamma_1){\wedge}\mathit{inv}(\Gamma_1) \quad I{=}\exists i{\in}1..p{\cdot}v_i{\in}\Theta_1\\ \Delta_3{=}\Delta_2{\circ_{\{n^*\}}}\rho_3\ \phi \quad \Theta_2{=}(\Theta_1{-}\bigcup_{i=1}^{p}\{v_i\}){\cup}\{v|I\}\\ (\neg I{\wedge}\Delta_3){\Rightarrow}\rho_3\ \phi_I \quad I{\Rightarrow}(A{=}\mathtt{U}) \quad Y{=}\bigcup_{i=1}^{p} V(t_i) \quad Z{=}Y{\cup}\mathit{prime}(Y)\end{array}}{\Gamma; C; \Delta; \Theta \vdash \mathtt{bind}\ v\ \mathtt{of}\ (v_1, \ldots, v_p)\ \mathtt{in}\ e :: t, \exists Z{\cdot}\Delta_3, \Theta_2}$$

$$[\underline{\textsc{con-ul}}]\quad \frac{\begin{array}{c}A_S{=}\mathit{ann}(t_S) \quad A_T{=}\mathit{ann}(t_T) \quad {\vdash}t_S{<:}t_T, \rho\\ \Theta_1{=}\Theta{\cup}(\{v\}{\lhd}A_S{=}\mathtt{U}{\wedge}A_T{\neq}\mathtt{L}{\rhd}\emptyset)\\ v{\notin}(\Theta{\cup}\Lambda) \quad \Lambda_1{=}\Lambda{\cup}(\{v\}{\lhd}A_T{=}\mathtt{L}{\rhd}\emptyset)\end{array}}{\Theta, \Lambda, \Psi \vdash \mathit{conUL}(v, t_S, t_T), \Theta_1, \Lambda_1, \Psi{\uplus}\rho}$$

$$[\underline{\textsc{call}}]\quad \frac{\begin{array}{c}\hat{t}_0\ \mathit{mn}((\hat{t}_i\ \hat{v}_i)_{i:1..p})\ \mathtt{where}\ \phi_{pr}; \phi_{po}\ \cdots \in P\\ t_0 = \mathit{fresh}(\hat{t}_0) \quad \Lambda_0 = \emptyset \quad \rho_0 = [\,] \quad \Gamma(v_i) = t_i \ \ i{\in}1..p\\ \Theta_{i-1}, \Lambda_{i-1}, \rho_{i-1} \vdash \mathit{conUL}(v_i, t_i, \hat{t}_i), \Theta_i, \Lambda_i, \rho_i \ \ i{\in}1..p\\ \rho = \mathit{rename}(\hat{t}_0, t_0){\uplus}\rho_p{\uplus}\mathit{prime}(\rho_p) \quad \Delta \approx\!\!>_{V(\Gamma)} \rho\phi_{pr}\\ X{=}\bigcup_{i=1}^{p} V(\hat{t}_i) \quad Y{=}X{\cup}\mathit{prime}(X) \quad L{=}\bigcup_{i=1}^{p} V(t_i)\end{array}}{\Gamma; C; \Delta; \Theta_0 \vdash \mathit{mn}(v_{1..p}) :: t_0, \Delta \circ_L \exists Y \cdot \rho\ \phi_{po}, \Theta_p}$$

$$[\underline{\textsc{meth}}]\quad \frac{\begin{array}{c}\Gamma = \{v_1{::}t_1, \ldots, v_p{::}t_p\} \quad \Delta{=}\mathit{no}\mathcal{X}(\Gamma){\wedge}\phi_{pr}{\wedge}\mathit{inv}(\Gamma)\\ \Gamma; C; \Delta; \emptyset \vdash e :: t, \Delta_1, \Theta \quad \vdash t{<:}t_0, \rho\\ \mathit{ann}(t_0) \neq \mathtt{L} \quad \mathit{ann}(t_i) = A_i, \ \ i{\in}0..p\\ Y = \bigcup_{i=1}^{p}(X_N{\cup}(X_T{\lhd}A_i{=}\mathtt{U}{\rhd}\emptyset) \mid (\_, X_T, X_N){=}\mathbf{V}_{\mathit{field}}(t_i))\\ (\exists\, \mathit{prime}(Y) \cdot \Delta_1){\Rightarrow}\rho\ \phi_{po} \quad \forall i{\in}1..p \cdot (A_i{=}\mathtt{L}){\Rightarrow}v_i \notin \Theta\end{array}}{C \vdash_{\mathit{meth}} t_0\ \mathit{mn}((t_i\ v_i)_{i:1..p})\ \mathtt{where}\ \phi_{pr}; \phi_{po}\ \{e\}}$$

$$[\underline{\textsc{if}}]\quad \frac{\begin{array}{c}\Gamma; C; \Delta{\wedge}b'{=}1; \Theta \vdash e_1 :: t_1, \Delta_1, \Theta_1\\ \Gamma; C; \Delta{\wedge}b'{=}0; \Theta \vdash e_2 :: t_2, \Delta_2, \Theta_2 \quad t{=}\mathit{unify}(t_1, t_2)\\ \Gamma(v){=}\mathtt{bool}\langle b\rangle\mathtt{@S} \quad \rho_i{=}\mathit{rename}(t_i, t), \ i = 1, 2\\ \Delta_3 = (\rho_1\Delta_1){\vee}(\rho_2\Delta_2) \quad \Theta_3 = \Theta_1 \cup \Theta_2\end{array}}{\Gamma; C; \Delta; \Theta \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 :: t, \Delta_3, \Theta_3}$$

$$[\underline{\textsc{subt}}]\quad \frac{\begin{array}{c}\mathbf{V}_{\mathit{obj}}(\tau\langle s_1, \ldots, s_m\rangle) = (S_I, S_T, \_)\\ (A_3, A_4){=}[\mathtt{R}{\mapsto}\mathtt{S}](A_1, A_2) \quad A_3{\leq_a}A_4\\ \rho_I = [n_i \mapsto s_i]_{s_i \in S_I} \quad \rho_T = [n_i \mapsto s_i]_{s_i \in S_T}\\ \rho = (\rho_I{\lhd}A_3 = \mathtt{S} \vee A_4 = \mathtt{S}{\rhd}\rho_I \uplus \rho_T)\end{array}}{\vdash\ \tau\langle s_1, \ldots, s_m\rangle\mathtt{@}A_1 <: \tau\langle n_1, \ldots, n_m\rangle\mathtt{@}A_2, \rho}$$

Figure 2. Major Size and Alias Type Rules for Safety Verification

5.2 Assignment

The assignment rule [ASSIGN] ensures that the LHS and RHS are consistent in accordance with (i) alias subtyping (ii) flow of trackable size properties (iii) imperative update of its LHS. It also checks that the LHS is neither a parameter variable, nor in R- or L-mode. To capture imperative update, we link the current size constraint with a new updated state for the affected size variables from the LHS. Given $\Gamma = \{\mathtt{x} :: \mathtt{int}\langle\mathtt{a}\rangle\mathtt{@S}\}$, we have:

$$\frac{\begin{array}{c}\neg\mathit{isParam}(\mathtt{x}) \quad \mathtt{S} \notin \{\mathtt{L}, \mathtt{R}\}\\ \Gamma; \Delta_1; \Theta \vdash \mathtt{x} + 1 :: \mathtt{int}\langle\mathtt{r}_1\rangle\mathtt{@S}, \Delta_2, \ \Theta\\ \Delta_2 = (\Delta_1{\wedge}\mathtt{r}_1{=}\mathtt{a}'{+}1) \quad \Gamma \vdash \mathtt{x} :: \mathtt{int}\langle\mathtt{r}_2\rangle\mathtt{@S}, \mathtt{a}' = \mathtt{r}_2, \{\mathtt{a}\}\\ \vdash \mathtt{int}\langle\mathtt{r}_1\rangle\mathtt{@S} <: \mathtt{int}\langle\mathtt{r}_2\rangle\mathtt{@S}, [\mathtt{r}_2 \mapsto \mathtt{r}_1]\end{array}}{\Gamma; C; \Delta_1; \Theta \vdash \mathtt{x} = \mathtt{x} + 1 :: \mathtt{void}\langle\rangle\mathtt{@S}, \exists\mathtt{r}_1{\cdot}\Delta_2 \circ_{\{\mathtt{a}\}} \mathtt{a}' = \mathtt{r}_1, \ \Theta}$$

If $\Delta_1 = (\mathtt{a}'{=}\ \mathtt{a})$, the final constraint simplifies to $\mathtt{a}'{=}\mathtt{a}{+}1$ which reflects the update. Apart from alias subtyping, the subtype relation [SUBT] also captures the flow of *trackable* size variables as a mapping. In the case of the above, when an S-object flows to an S-location, only the size-immutable size variable is captured via its mapping, namely $[\mathtt{r}_2 \mapsto \mathtt{r}_1]$. Some other examples are:

$$\begin{array}{l}\vdash \mathtt{Pair}\langle\mathtt{s}_1, \mathtt{s}_2, \mathtt{s}_3\rangle\mathtt{@U} <: \mathtt{Pair}\langle\mathtt{n}_1, \mathtt{n}_2, \mathtt{n}_3\rangle\mathtt{@S}, [\mathtt{n}_2 \mapsto \mathtt{s}_2]\\ \vdash \mathtt{Pair}\langle\mathtt{s}_1, \mathtt{s}_2, \mathtt{s}_3\rangle\mathtt{@U} <: \mathtt{Pair}\langle\mathtt{n}_1, \mathtt{n}_2, \mathtt{n}_3\rangle\mathtt{@U}, [\mathtt{n}_i \mapsto \mathtt{s}_i]_{i=1}^{3}\\ \vdash \mathtt{Pair}\langle\mathtt{s}_1, \mathtt{s}_2, \mathtt{s}_3\rangle\mathtt{@S} <: \mathtt{Pair}\langle\mathtt{n}_1, \mathtt{n}_2, \mathtt{n}_3\rangle\mathtt{@U}, \quad \textbf{!FAIL}\end{array}$$

Another important aspect of assignment is the restoration of the uniqueness for the LHS variable/field. As this is not applicable to shared objects, let us examine another example with $\Delta_2 = \mathit{no}\mathcal{X}\{\mathtt{d}\}$. Notice a smaller dead set.

$$\frac{\begin{array}{c}\neg\mathit{isParam}(\mathtt{v}) \qquad \mathtt{U} \notin \{\mathtt{L}, \mathtt{R}\}\\ \Gamma; \Delta_2; \{\mathtt{v}\} \vdash \mathtt{new}\ \mathtt{Int}(4) :: \mathtt{Int}\langle\mathtt{r}_1\rangle\mathtt{@U}, \Delta_2{\wedge}\mathtt{r}_1{=}4, \{\mathtt{v}\}\\ \Gamma \vdash \mathtt{v} :: \mathtt{Int}\langle\mathtt{r}_2\rangle\mathtt{@U}, \mathtt{d}'{=}\mathtt{r}_2, \ \{\mathtt{d}\}\\ \vdash \mathtt{Int}\langle\mathtt{r}_1\rangle\mathtt{@U} <: \mathtt{Int}\langle\mathtt{r}_2\rangle\mathtt{@U}, [\mathtt{r}_2 \mapsto \mathtt{r}_1]\end{array}}{\Gamma; C; \Delta_2; \{\mathtt{v}\} \vdash \mathtt{v} = \mathtt{new}\ \mathtt{Int}(4) :: \mathtt{void}\langle\rangle\mathtt{@S}, \mathtt{d}'{=}4, \ \{\}}$$

5.3 Conditional

For conditional, our rule is able to track the size constraint path-sensitively, but the dead set is tracked only flow-sensitively but not path-sensitively. Given $\Gamma_1{=}\ \Gamma{+}\{\mathtt{x}{::}\mathtt{bool}\langle\mathtt{h}\rangle\mathtt{@S}, \mathtt{v}{::}\mathtt{Int}\langle\mathtt{j}\rangle\mathtt{@U}\}$ and $\Delta{=}(\mathtt{h}'{=}0)$, we have:

$$\frac{\begin{array}{c}\Gamma_1;C;\Delta\wedge \mathtt{h}'=1;\{\}\vdash \mathtt{v}::\mathtt{Int}\langle \mathtt{r}_1\rangle@\mathtt{U},\mathtt{false},\{\mathtt{v}\}\\ \Gamma_1;C;\Delta\wedge \mathtt{h}'=0;\{\}\vdash \mathtt{new\ Int}(5)::\mathtt{Int}\langle \mathtt{r}_2\rangle@\mathtt{U},\Delta\wedge \mathtt{r}_2=5,\{\}\end{array}}{\begin{array}{c}\Gamma_1;C;\Delta;\{\}\vdash \mathtt{if\ x\ then\ v\ else\ (new\ Int(5))}\\ \qquad :: \mathtt{Int}\langle \mathtt{r}_3\rangle@\mathtt{U},\Delta\wedge \mathtt{r}_3=5,\ \{\mathtt{v}\}\end{array}}$$

Note that $\mathtt{h}'=1$ and $\mathtt{h}'=0$ captures path-sensitivity. As a result, the final size constraint is able to ignore an infeasible branch. A branch is infeasible if its corresponding size constraint yields `false`. However, the uniqueness of `v` is consumed even though it occurs in the infeasible branch. This is so as the dead set is tracked only flow-sensitively.

5.4 Method Declaration and Call

For method declaration, our type rule must ensure that each decoration at the method header is consistent with its body. The declared size constraint may be weaker than the analysed size constraint (from the method body), but not vice versa. Note that non-trackable size variables and mutable size variables of unique parameters must be quantified, as we do not track them across method boundary.

The rule for method call must ensure that `U`-mode parameters consume uniqueness, and that `L`-mode parameters adhere to the lent-once policy. Relation $\Theta,\Lambda,\Psi\vdash \mathit{conUL}(v,t_S,t_T),\Theta_1,\Lambda_1,\Psi_1$ (named [CON−UL]) helps ensure the above. It checks that each variable v is neither lent twice, nor has its uniqueness consumed twice. Note that Λ captures unique variables which are temporarily lent out. The method invocation rule also includes a safety check on the precondition of the callee. This is performed by the predicate $\approx\!\!>_X$ defined as follows:

$$\Delta \approx\!\!>_X \phi =_{df} (\Delta\Rightarrow\rho\phi),\ \text{where}$$
$$\rho=[s_1\mapsto s_1',..,s_n\mapsto s_n']\ \text{and}\ V_u(\phi)\cap X=\{s_1,..,s_n\}.$$

where V_u returns the size variables in unprimed form, for example $V_u(x'{=}z{+}1\wedge y{=}2)=\{x,y,z\}$. As an example of checking a method call, given

$$\Gamma=\{\mathtt{p}::\mathtt{Buffer}\langle \mathtt{r},\mathtt{z}\rangle@\mathtt{U},\mathtt{v}::\mathtt{int}\langle \mathtt{m}\rangle@\mathtt{S}\}$$
$$\Delta=(\mathtt{r}'{=}3\wedge \mathtt{r}{=}0\wedge \mathtt{z}{=}10\wedge \mathtt{m}{=}30\wedge \mathit{no}\mathcal{X}\{\mathtt{z},\mathtt{m}\})$$

we obtain $V(\Gamma)=\{\mathtt{r},\mathtt{z},\mathtt{m}\}$ and the following:

$$\frac{\dfrac{\begin{array}{c}t_1=\mathtt{Buffer}\langle \mathtt{r},\mathtt{z}\rangle@\mathtt{U}\quad \hat{t}_1=\mathtt{Buffer}\langle \mathtt{s},\mathtt{c}\rangle@\mathtt{L}\\ \mathtt{p}\notin\Theta\quad \rho_1=[\mathtt{s}\mapsto\mathtt{r},\mathtt{c}\mapsto\mathtt{z}]\end{array}}{\Theta,\emptyset,[\,]\vdash \mathit{conUL}(\mathtt{p},t_1,\hat{t}_1),\Theta,\{\mathtt{p}\},\rho_1}\quad \dfrac{\begin{array}{c}t_2=\mathtt{int}\langle \mathtt{m}\rangle@\mathtt{S}\quad \hat{t}_2=\mathtt{int}\langle \mathtt{n}\rangle@\mathtt{S}\\ \mathtt{v}\notin\Theta\cup\{\mathtt{p}\}\quad \rho_2=\rho_1\uplus[\mathtt{n}\mapsto\mathtt{m}]\end{array}}{\Theta,\{\mathtt{p}\},\rho_1\vdash \mathit{conUL}(\mathtt{v},t_2,\hat{t}_2),\Theta,\{\mathtt{p}\},\rho_2}\quad \Delta\approx\!\!>_{V(\Gamma)}(\mathtt{r}<\mathtt{z})}{\Gamma;C;\Delta;\Theta\vdash \mathtt{add}(\mathtt{p},\mathtt{v})::\mathtt{void}\langle\rangle@\mathtt{S},\Delta_2,\ \Theta}$$

Note that uniqueness of `p` is only lent-out and not consumed. We also have:

$$\Delta_2=\Delta\circ_{\{\mathtt{r},\mathtt{z},\mathtt{m}\}}(\mathtt{r}'=\mathtt{r}+1\wedge \mathit{no}\mathcal{X}\{\mathtt{z},\mathtt{m}\})$$
$$=(\mathtt{r}'{=}4\wedge \mathtt{r}{=}0\wedge \mathtt{z}{=}10\wedge \mathtt{m}{=}30\wedge \mathit{no}\mathcal{X}\{\mathtt{z},\mathtt{m}\})$$

5.5 Binding

The **bind** expression enables us to accurately capture the size relations among object fields as well as between an object and its fields. In addition, it can replace the field access construct altogether (for type checking). When an object subjected to a **bind** operation is marked by either `R` or `S`, all its unique fields are remarked as `S` using $[\mathtt{U}\mapsto\mathtt{S}]$ in the scope of **bind**. On the other hand, the object will lose its uniqueness if any one of its fields loses it. For instance, given $\Gamma{=}\Gamma_0{+}\{\mathtt{v}::\mathtt{List2}\langle \mathtt{n}\rangle@\mathtt{U}\}$, the following derivation demonstrates how `v` loses its uniqueness when its unique `y` field loses it (denoted by $I=(\mathtt{y}{\in}\Theta_1)$):

$$\frac{\begin{array}{c}\mathtt{v}\notin\Theta\quad \Gamma_0\cup\Gamma_1;C;\Delta_1;\Theta\vdash \mathtt{y}::t_1,\Delta_2,\Theta_1\quad \Theta_1=\Theta\cup\{\mathtt{y}\}\\ \Gamma(\mathtt{v}){=}t\quad A{=}\mathit{ann}(t)\quad I{=}(\mathtt{y}{\in}\Theta_1)\quad I{\Rightarrow}A{=}\mathtt{U}\quad \neg I{\wedge}\Delta_3{\Rightarrow}\mathtt{n}'{\geq}0\end{array}}{\Gamma;C;\Delta;\Theta\vdash \mathtt{bind\ v\ of\ (x,y)\ in\ y}::t_1,\Delta_4,\Theta_1{-}\{\mathtt{y}\}{\cup}\{\mathtt{v}|I\}}$$

for some values $\Gamma_1, t_1, \Delta_1,\Delta_2, \Delta_3$ and Δ_4. As a counter-example, consider `bind v of (x, y) in x`. Here, `v` does not lose its uniqueness as the uniqueness of its `y` field remains intact.

A well-typed **bind** expression also permits the size invariant ϕ_I of the bound object to be *temporarily violated* in its scope, but ensures that ϕ_I holds whenever the object escapes. This is enforced by the check $(\neg I{\wedge}\Delta_3){\Rightarrow}\rho_3\ \phi_I$ with I denoting the loss of one or more unique fields. Our rule also ensures that an object and its unique fields are never used simultaneously to avoid aliasing. It achieves this by hiding the visibility of variable v through $\Gamma{-}\{v|A{\in}\{\mathtt{L},\mathtt{U}\}\}$ when type-checking the body of the **bind** construct.

5.6 Soundness of Type System

We have proposed a small-step operational semantics instrumented with alias and size notations. We have also formulated and proved several novel safety properties that our checking system possesses:

- **Alias Property**: (1) All unique references are unaliased during the evaluation; (2) each unique reference can only be lent once within each stack frame; and (3) `R`-mode fields never change during evaluation.
- **Type Preservation**: Each well-typed program preserves its type under reduction with a runtime environment and a store which are consistent with the compile-time counterparts. The final size property is consistent with the value obtained on termination.
- **Progress**: Well-typed programs can never go wrong. It guarantees that safety policies are met for well-typed OIMP programs.

Details on the dynamic semantics, formal descriptions of the soundness theorems and their proofs can be found in our technical report [11]. Detailed correctness proof for size analysis can also be found in [31] in the context of a first-order functional language with primitive resources.

6. Related Work

Sized type was first proposed by Hughes, Pareto and Sabry [23, 22] as a formal mechanism for proving the correctness of certain size-related properties for embedded functional programs. Their proposal captures an upper or lower bound for each data, depending on whether it is declared as data or co-data. This catered to independent attribute analysis, rather than relational analysis. Separately, dependent type was proposed by Xi and Pfenning [36, 38]

Programs	Size (lines)		Safety Checking
	Source	Annotation	(seconds)
bisort	340	7	0.68
em3d	462	19	0.30
health	562	22	6.08
mst	473	31	0.20
power	765	24	1.32
treeadd	195	6	0.12
tsp	545	10	0.42
perimeter	745	12	8.48
n-body	1128	31	0.37
Voronoi	1000	45	1.90
insert(AVL)	64	1	1.36
delete(AVL)	157	2	4.79
bs(array)	41	1	0.08
buffer	65	6	0.12

Figure 3. Performance of Verification

to capture size information and has been applied to a variety of applications. These early works are based on the type checking framework. Later, Chin and Khoo [10] enriched sized typing by devising fixed-point methods for safely inferring the size relations for functional programs. A common feature of these past works is that they are based primarily on functional languages, and operate over mostly immutable data structures.

Recently, some advance has been achieved in extending dependent types to imperative programming through a language, called Xanadu [37]. However, Xanadu supports only a limited set of types, including read-only lists; but not general objects. This restriction avoided the aliasing problem, as only *size-immutable* objects are effectively used. With a different research goal, Odersky et. al. [29] designed a dependent type system for objects and classes. However, its purpose is to model JAVA's inner classes, virtual types and family polymorphism, rather than tracking size properties.

Sharing objects through aliases is both a powerful feature and a weakness of object-oriented languages [1, 7, 19]. As each object may be updated via any alias, object being changed may not even be aware of it; causing such programs to be harder to understand and reason. To regain control over aliasing, various researchers have proposed alias annotations to limit the capability of pointer variables so that local reasoning is possible. A wide range of alias annotations have been proposed, which have been systematically organised into a general capability system[1, 7]. Our work is built on top of such alias controls. Uniqueness [27] gives unaliased reference, and is closely related to linear type systems [35, 17]. Lent mechanism allows other annotations to be borrowed temporarily over a program scope, where they do not escape. It is related to quasi-linearity [24]. Read-only annotation is related to immutability, and can facilitate sharing and invariants. Another well-studied mechanism is ownership annotation in support of object encapsulation. Here, every object has an owner, whose access is limited to the ownership tree, such that external objects have only indirect accesses through the owner's methods. Ownership encapsulation has been applied to a number of applications, including region-types for real-time Java [4], prevention of data races and deadlocks in concurrent programs[5, 6], and as aids to program understanding [1]. Our present proposal for sized type for objects rely on a smaller set of alias annotations, but offers a mutually beneficial synergy. With the portfolio of size-inspired applications, we expect the type system proposed here to further reinforce on the importance of alias control mechanisms; and to continue to benefit from their advances.

Instead of confining aliasing[19], there have been a number of attempts at analysing possible aliasing structures. Systems such as TVLA [33], PALE [28], and Roles [25], enable the shape of local object graphs to be analysed and captured. Most of these systems employ heavier machinery to capture state change and to enforce the expected aliasing structures. For example, Sagiv, Rep and Wilhelm [33] employed three-value logic to capture points-to relation between objects as well as imperative changes on shape graph. These proposals are complimentary to our goal of size analysis in relational form, and may open up new classes of mutable objects that could be precisely analysed.

In recent years, several approaches have been advocated for verifying software to ensure that it conforms to stated safety policy. One popular approach [20, 2, 12, 8] is based on the concept of model checking in which program states are approximated through boolean predicates. Choosing a suitable predicate abstraction remains one of the main challenges, and various symbolic techniques have been proposed to facilitate this task. With a different approach, several type based techniques [14, 16, 13] have also been advocated for checking if user programs conform to some stated safety protocols. The focus has been on designing bounded typestates together with linear typing for supporting strong updates. As an approach between the above two, Flanagan et.al.[15] use verification-condition generation and automatic theorem-proving techniques in their extended static checking system for Java. However, they incorporate a trade-off between soundness and usefulness. Moreover, most of these proposals are based on finite state approximation of the program states, and are typically formulated in a path-insensitive manner.

Through the use of relational size analysis (with disjunctive formulae), our proposal has the potential to give more precise verification. Furthermore, our use of alias controls and size-immutability (often omitted in other works) has provided a clearer path towards analysing object-based imperative programs.

7. Conclusion

We have proposed an advanced type system to capture size relations for objects. We make use of size-immutability to facilitate sharing, and alias controls to identify unaliased objects. This mechanism permits size relation in constraint form to be accurately analysed for object-based programs. Our type system can be viewed as a sound and decidable lightweight verification method. We use the ADT mechanism augmented with size invariants together with pre/post conditions on methods, to allow us to express a variety of size-inspired safety policies - ranging from safe array bounds, data structure invariant to protocol verification.

We have implemented a prototype system which converts annotated Java programs (without concurrency and exceptions) to our kernel language, before type checking is applied to verify the specified safety policies (with a prototype system accessible at `http://loris-4.ddns.comp.nus.edu.sg/~nguyenh2/mi`). Our prototype was implemented using the Glasgow Haskell compiler. To evaluate this prototype, we hand-annotated a set of programs from the Java Olden benchmark together with programs described in this paper, before subjecting them to sized type checking. The performance statistics (on Red Hat Linux 9.0 for Pentium 2.4 GHz with 768MB) is summarised in Fig 3. Due to modular type checking, our system verifies each medium-sized program under 10 seconds, despite the use of Presburger constraint solving[32]. Size and alias annotations are required for the headers of object and method declarations which account for less than 5% of the source code. Our recent investigations into automatic inference of size constraints for an imperative language without objects [39] (for array bound check elimination) could further reduce the burden of annotations.

Acknowledgments

The authors would like to thank Florin Craciun, Dana Xu, Stefan Andrei and Martin Rinard for various pointers, and the referees of ICSE05 for their insightful comments. This work is supported by Singapore-MIT Alliance and NUS grant R252-000-092-112.

8. REFERENCES

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotation for Program Understanding. In *ACM OOPSLA*, Seattle, Washington, November 2002.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *ACM PLDI*, June 2001.

[3] F. Benoy and A. King. Inferring argument size relationships with CLP(R). In *Logic Programming Synthesis and Transformation*, Springer-Verlag, August 1997.

[4] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM OOPSLA*, Seattle, Washington, November 2002.

[6] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *ACM OOPSLA*, Tampa Bay, Florida, October 2001.

[7] J. Boyland, J. Noble, and W. Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *ECOOP*, Budapest, Hungary, June 2001.

[8] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In *Proceedings of the International Conference on Software Engineering (ICSE'03)*, pages 385–395, 2003.

[9] E. C. Chan, J. Boyland, and W. L. Scherlis. Promises: Limited Specifications for Analysis and Manipulation. In *Proceedings of the International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.

[10] W.N. Chin and S.C. Khoo. Calculating sized types. In *ACM SIGPLAN PEPM*, pages 62–72, Boston, Massachusetts, United States, January 2000.

[11] W.N. Chin, S.C. Khoo, S.C. Qin, C. Popeea, and H.H. Nguyen. Verifying Safety Policies with Size Properties and Alias Controls. Technical report, SoC, Natl Univ. of Singapore, August 2004. avail. at http://www.comp.nus.edu.sg/~qinsc/papers/safety.ps.gz.

[12] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of the International Conference on Software Engineering (ICSE'00)*, pages 439–448, Limerick, Ireland, June 2000.

[13] R. DeLine and M. Fahndrich. Typestates for Objects. In *ECOOP*, Oslo, Norway, June 2004.

[14] M. Fahndrich and R. DeLine. Adoption and Focus: Practical linear types for imperative programming. In *ACM PLDI*, June 2002.

[15] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended Static Checking for Java. In *ACM PLDI*, Berlin, Germany, June 2002.

[16] J.S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *ACM PLDI*, Berlin, Germany, June 2002.

[17] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[18] B. Grobauer. Cost Recurrences for DML Programs. In *Proceedings of the International Conference on Functional Programming (ICFP '01)*, pages 253–277, Florence, Italy, September 2001.

[19] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating Objects with Confined Types. In *ACM OOPSLA*, October 2001.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *ACM POPL*, pages 58–70, Portland, Oregon, January 2002.

[21] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First Order Functional Programs. In *ACM POPL*, New Orleans, Louisiana, January 2003.

[22] J. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space: Towards Embedded ML Programming. In *Proceedings of the International Conference on Functional Programming (ICFP '99)*, September 1999.

[23] J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *ACM POPL*, pages 410–423. ACM Press, January 1996.

[24] N. Kobayashi. Quasi-linear types. In *ACM POPL*, pages 29–42. ACM Press, 1999.

[25] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *ACM POPL*, Portland, Oregon, January 2002.

[26] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *ACM POPL*, pages 81–92. ACM Press, 2001.

[27] N. Minsky. Towards alias-free pointers. In *ECOOP*, July 1996.

[28] A. Moeller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *ACM PLDI*, June 2001.

[29] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In *ECOOP*, Springer LNCS, July 2003.

[30] B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997.

[31] C. Popeea and W.N. Chin. A Type System for Resource Protocol Verification and its Correctness Proof. In *ACM SIGPLAN PEPM*, Verona, Italy, August 2004.

[32] W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.

[33] M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. In *ACM POPL*, January 1999.

[34] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes (extended abstract). In *Proceedings of the 10th ACM Symposium on Principles of Database Systems*, pages 216–226, 1991.

[35] P. Wadler. Linear types can change the world! In *M. Broy and C. Jones, editors, Programming Concepts and Methods*. North Holland, Amsterdam, April 1990.

[36] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[37] H. Xi. Imperative Programming with Dependent Types. In *Proceedings of 15th Symposium on Logic in Computer Science (LICS '00)*, Santa Barbara, June 2000.

[38] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM PLDI*, pages 249–257. ACM Press, June 1998.

[39] D.N. Xu, C. Popeea, S.C. Khoo, and W.N. Chin. Scalable and Precise Inference for Array Checks Elimination. Technical report, SoC, Natl Univ. of Singapore, November 2004. avail. at http://www.comp.nus.edu.sg/~chinwn/publ.html.