On Role Logic

Viktor Kuncak and Martin Rinard
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{vkuncak,rinard}@csail.mit.edu

MIT CSAIL Technical Report No 925
Internal Manuscript VK0101, October 2003

Abstract

We present *role logic*, a notation for describing properties of relational structures in shape analysis, databases, and knowledge bases. We construct role logic using the ideas of de Bruijn's notation for lambda calculus, an encoding of first-order logic in lambda calculus, and a simple rule for implicit arguments of unary and binary predicates.

The unrestricted version of role logic has the expressive power of first-order logic with transitive closure. Using a syntactic restriction on role logic formulas, we identify a natural fragment RL^2 of role logic. We show that the RL^2 fragment has the same expressive power as two-variable logic with counting C^2, and is therefore decidable.

We present a translation of an imperative language into the decidable fragment RL^2, which allows compositional verification of programs that manipulate relational structures. In addition, we show how RL^2 encodes boolean shape analysis constraints and an expressive description logic.

Keywords: Program Verification, Shape Analysis, Static Analysis, Two-Variable Logic with Counting, Description Logic, First-Order Logic, Types, Roles, Object-Models

Draft of October 31, 2003, 1:37pm,
see http://www.mit.edu/~vkuncak/papers for later versions.

Contents

1 Introduction

Systems as relational structures. Complex systems arising in many areas of Computer Science can be naturally represented as relational structures. The state of an imperative program can be specified using sets and relations denoted by unary and binary predicates [24, 32, 66, 8], especially for object-oriented programs [36, 63]; a relational database is a finite relational structure [18, 16]; knowledge bases and deductive databases can also be based on predicate logic [1, 41, 53].

Shape analysis. Shape analysis techniques [65, 29, 33, 26, 27, 25, 17, 40, 39, 43, 37, 55] can verify and derive precise properties of objects in the heap. Shape analysis is therefore important for reasoning about programs written in modern imperative programming languages. Shape analysis is also promising as a general-purpose verification technique, because of its ability to reason about graphs as general structures, and the ability to summarize properties of unbounded sets of objects.

Many of the shape analysis techniques have a logical foundation: [65] is based on (two-valued and three-valued) first-order logic with transitive closure, [39, 40, 37, 55] is based on monadic second-order logic of trees, [26, 27] is based on graph grammars which are closely related to monadic second-order logic of trees [62]. Theorem proving is used in [33] to derive consequences of axioms about data structures. Many shape analyses perform abstract interpretation [19] to synthesize loop invariants [65, 29, 43].

Role logic. This paper presents *role logic*, a notation for describing properties of relational structures in shape analysis, databases, and knowledge bases. Role logic is an attempt to simultaneously achieve the simplicity of the role declarations of [43] with a transparent connection with the well-established first-order logic.

On the one hand, the full role logic has the expressive power of first order logic with transitive closure, which makes it as expressive as the logic of [65, 36] and more expressive than the original role constraints [43]. For example, role logic is closed under all propositional operations and generalizes boolean shape analysis constraints [48]. Role logic formulas easily translate into the traditional first-order logic notation.

On the other hand, like the specialized notation for declaring roles in [43], role logic allows natural description of common properties of imperative data structures with mutable references. Like dynamic logics [31] and description logics [1], role logic allows suppressing names of variables, which often leads to concise specifications. The conciseness of role logic makes it an appealing choice for lightweight annotations in a programming language.

Another property that role logic shares with description logics is that an interesting subset of role logic is decidable. We show the decidability of the fragment RL^2 of role logic in Section 4 by establishing a correspondence with the two-variable logic with counting C^2 [30, 57]. While many description logics are known to be representable in C^2 but are potentially weaker than C^2, the fragment RL^2 of role logic matches precisely the expressive power of C^2.

Contributions. The following are the main contributions of this paper:

1. We introduce *role logic*, which applies the ideas of implicit arguments and deBruijn's lambda calculus notation to first order logic (Section 3). The result is a concise way of specifying properties of first-order structures that arise in shape analysis, databases, and knowledge bases.

2. We define a variable-free subset RL^2 of role logic (Section 4). We give a translation of RL^2 formulas to formulas of two-variable logic with counting C^2. This translation implies that RL^2 is decidable, because C^2 is decidable [30]. We further give a translation of C^2 formulas to RL^2 formulas. These two translations imply that RL^2 is just as expressive as C^2.

3. As the main application of role logic, in Section 5.1 we present a compositional shape analysis technique. We introduce a unified language for writing implementations, specifications, and conformance claims. The constructs of the language denote relations on program states expressible in the decidable fragment RL^2. The analysis technique is based on generating verification conditions in RL^2 and applying the decision procedure for RL^2. The analysis verifies the correctness of the dynamically changing referencing relationships between objects by showing that procedures conform to their specifications. By conjoining procedure specifications with global invariants, the analysis can also show that the program preserves the key data structure consistency properties necessary for the correct execution of the program.

4. We present two additional applications of role logic:

   (a) we show in Section 5.3 that a subset of role logic RL^2 naturally corresponds to an expressive description logic [1, Chapter 5];

   (b) we note in Section 5.2 that boolean shape analysis constraints [48], which can describe the basic structure of data-flow facts in [65], are a subset of constraints expressible in role logic.

2 Example

To give a flavor of role logic, we present an example that illustrates one aspect of a client-server manager system that assigns clients to servers. Figure 1 is a standard object model that graphically displays the system, using boxes to represent sets, arrows to represent relations, and intervals $N..M$ to represent constraints on relations. Figure 2 describes the same system using role logic. Figure 3 presents a fragment of the code of the system. The code is expressed in an imperative language extended with specification constructs.

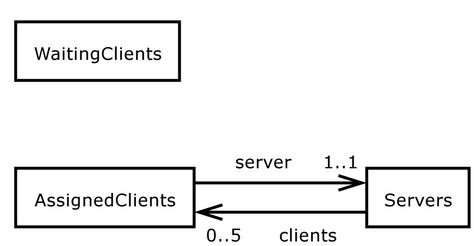

Figure 1: An object model for a component of client-server manager

$$
\begin{aligned}
&\mathsf{GlobalInvariant} = \\
&\quad \{\mathsf{Servers}\} \;\wedge\; (\mathsf{disjoint}\,\mathsf{Servers}, \mathsf{Clients}) \wedge \\
&\quad (\mathsf{partition}\,\mathsf{Clients};\; \mathsf{WaitingClients}, \mathsf{AssignedClients}) \wedge \\
&\quad [[\mathsf{server} \Rightarrow \mathsf{AssignedClients}' \wedge \mathsf{Servers}]] \wedge \\
&\quad [[\mathsf{clients} \Leftrightarrow {\sim}\mathsf{server}]] \wedge \\
&\quad [\mathsf{AssignedClients} \Rightarrow \mathsf{card}^{=1}\mathsf{server}] \wedge \\
&\quad [\mathsf{Servers} \Rightarrow \mathsf{card}^{\leq 5}\mathsf{clients}]
\end{aligned}
$$

Example consequence:

$$
\begin{aligned}
P \;\equiv\; &[\mathsf{WaitingClients} \Rightarrow \\
&\quad [\neg(\mathsf{clients} \vee \mathsf{server} \vee {\sim}\mathsf{clients} \vee {\sim}\mathsf{server})]]
\end{aligned}
$$

Figure 2: Global constraints of the client-server manager, expressed in role logic

Global constraints. Figure 2 describes the global constraints of a client-server manager system using a conjunction of role logic formulas. There are two basic kinds of objects in the system: servers and clients. We model these objects using two disjoint sets $\mathsf{Clients}$ and $\mathsf{Servers}$. The set $\mathsf{Clients}$ is further partitioned into the set $\mathsf{AssignedClients}$ of objects that have been assigned to servers, and the set $\mathsf{WaitingClients}$ that have not been assigned yet. The $\mathsf{disjoint}$, $\mathsf{partition}$, and other constructs of set algebra of sets and relations ($\cap$, $\cup$, $\setminus$) are definable in role logic.

We require the set $\mathsf{Servers}$ to be non-empty, which we denote by $\{\mathsf{Servers}\}$, with the meaning $\exists x.\mathsf{Servers}(x)$. The constraint $[[\mathsf{server} \Rightarrow \mathsf{AssignedClients}' \wedge \mathsf{Servers}]]$ translates to $\forall x.\forall y.\, \mathsf{server}(x,y) \Rightarrow \mathsf{AssignedClients}(x) \wedge \mathsf{Servers}(y)$. Namely, the brackets [ ] corresponds to a universal quantifier. An occurrence of a binary predicate (such as server) is implicitly supplied with the previous-innermost bound variable (here, x) and the innermost bound variable (here, y). The occurrence of an unary predicate $\mathsf{Servers}$ is supplied with the innermost bound variable (y), unless the unary predicate is primed, in which case the previous-innermost bound variable (in this case x) is supplied instead. The constraint $[[\mathsf{clients} \Leftrightarrow {\sim}\mathsf{server}]]$ means that the relation $\mathsf{clients}$ is the inverse of the relation server. The constraint $[\mathsf{Servers} \Rightarrow \mathsf{card}^{\leq 5}\mathsf{clients}]$ translates into the formula $\forall x.\, \mathsf{Servers}(x) \Rightarrow \exists^{\leq 5} y.\, \mathsf{clients}(x,y)$ in first-order logic with counting quantifiers.

Note that all of our translations of constraints in Figure 2 use only two variables, x and y. In fact, our entire example is expressed in the RL^2 fragment of role logic. In Section 4 we show that RL^2 corresponds to the decidable fragment C^2 of two-variable first-order logic with counting, and is therefore decidable. Figure 2 presents the formula P denoting the fact that $\mathsf{WaitingClients}$ objects have no incoming or outgoing edges. If we apply the decision procedure for RL^2, we can show that $\mathsf{GlobalInvariant} \Rightarrow P$ is a valid formula, which means that P is a logical consequence of $\mathsf{GlobalInvariant}$. By querying whether the $\mathsf{GlobalInvariant}$ implies properties of interest such as P, the developers can increase their confidence in the correctness and completeness of the design. Moreover, our technique can be used to show the conformance of the program with respect to the design.

```
proc assignClients() =
spec old(GlobalInvariant) => !{WaitingClients} &
 [AssignedClients <=>
        old(AssignedClients | WaitingClients)] &
 GlobalInvariant

proc assignClientsIMPL() = {
  if ({WaitingClients}) {
    cl := getWaitingClient();
    assignOneClientIMPL(cl);
    assignClientsIMPL();
}}
claim: assignClientsIMPL => assignClients

proc assignOneClient(cl) =
spec old(GlobalInvariant &
     [cl => WaitingClients]) =>
 [WaitingClients | cl <=> old(WaitingClients)] &
 [AssignedClients <=> old(AssignedClients) | cl] &
 GlobalInvariant

proc assignOneClientIMPL(cl) = {
  sv := getServer();
  if (Card (sv' & clients) <= 4) {
    WaitingClients := WaitingClients \ cl;
    AssignedClients := AssignedClients | cl;
    cl.server := sv;
    sv.clients := sv.clients | cl;
  } else {
    assignOneClientIMPL(cl);
}}
claim: assignOneClientIMPL => assignOneClient

proc getWaitingClient() : set =
  spec {WaitingClients} =>
       skip & [returned => WaitingClients]

proc getServer() : set =
  spec {Servers} =>
       skip & [returned => Servers]
```

Figure 3: A fragment of a program that assigns $\mathsf{WaitingClients}$ to $\mathsf{Servers}$

Program fragment. Figure 3 shows a fragment of the code of the client-server manager. The top-level procedure in the code is a tail-recursive procedure `assignClientsIMPL` that processes all $\mathsf{WaitingClients}$ objects and assigns them to $\mathsf{Servers}$ objects. The `assignClientsIMPL` procedure terminates if there are no $\mathsf{WaitingClients}$ objects. Otherwise, it uses the `getWaitingClient` procedure to obtain an element of $\mathsf{WaitingClients}$ and assigns it to some $\mathsf{Servers}$ object using the `assignOneClient` procedure, and continues with the next $\mathsf{WaitingClients}$ object using a tail-recursive call.

The partial correctness of the procedure `assignClientsIMPL` is given using the specification `assignClients`. The requirement that the procedure conforms to its specification is stated using the construct

```
claim: assignClientsIMPL => assignClients
```

The verification of each procedure call site uses only procedure specification (summary) instead of the body of the procedure, which allows verification of recursive procedures. In this example, the implementations of procedures `getWaitingClient` and `getServer` are not available, which illustrates the advantage of assume/guarantee reasoning for partitioning a verification task.

Using the translation in Section 5.1, the `claim` constructs are reduced to verification conditions expressed in role logic. For a large class of constructs presented in Section 5.1, and our example in particular, the resulting verification conditions belong to the decidable RL^2 and can therefore be discharged using a decision procedure for RL^2.

Note that we are able to express detailed specifications of the correctness of procedures while remaining in the decidable logic. For example, the specification `assignClients` ensures that the entire global invariant in Figure 2 is preserved, and that no client objects are lost in the assignment process: after `assignClients`, the set $\mathsf{AssignedClients}$ is the union of the old value of $\mathsf{AssignedClients}$ and the old value of $\mathsf{WaitingClients}$, whereas the new value of $\mathsf{WaitingClients}$ is an empty set.

3 A Recipe for Role Logic

In this section we motivate the role logic by constructing it in several steps. We start with first-order logic encoded in the simply typed lambda calculus; we then move to the notation that refers to each variable by its index. Finally, we impose a rule for implicitly supplying the indices of variables to predicate symbols. Later, in Section 3.6, we summarize the syntax and the semantics of role logic, and in Section 4 we present a decidable sublogic of role logic.

3.1 Lambda Calculus

Figure 4 presents simply typed lambda calculus with explicit type annotations in lambda abstraction (the Church-style simply typed lambda calculus [5, Section 3.2]). This calculus is our starting point.

As primitive types we use bool for boolean values, and obj for objects. As the only type constructor we use arrow $\rightarrow$. We introduce rel^k as a shorthand type defined by

$$\begin{array}{rcl} \mathsf{rel}^0 & \equiv & \mathsf{bool} \\ \mathsf{rel}^{k+1} & \equiv & \mathsf{obj} \rightarrow \mathsf{rel}^k \end{array}$$

Simple types enable us to give a simple set-theoretic semantics to formulas by interpreting lambda abstractions as total functions. The resulting semantics is in Figure 4; the semantics is straightforward because we use lambda calculus itself as our meta-notation.

$$\begin{array}{rcll} \mathsf{Form} & = & \mathsf{Vars} & \text{variable lookup} \\ & & & \mathsf{Vars} = \{x, f, \ldots\} \\ & | & \mathsf{Form}\ \mathsf{Form} & \text{function application} \\ & | & \lambda\mathsf{Vars} : \mathsf{Type}.\mathsf{Form} & \text{function abstraction} \end{array}$$

Syntax

$$\frac{\Gamma(v) = T}{\Gamma \vdash v : T}$$

$$\frac{\Gamma \vdash F_1 : T_1 \rightarrow T_2, \quad \Gamma \vdash F_2 : T_1}{\Gamma \vdash F_1 F_2 : T_2}$$

$$\frac{\Gamma[v := T_1] \vdash F : T_2}{\Gamma \vdash (\lambda v : T_1.F) : T_1 \rightarrow T_2}$$

Types

$$\begin{array}{rcl} [\![v]\!]\, e & = & e\, v \\ [\![F_1\ F_2]\!]\, e & = & ([\![F_1]\!] e)\ ([\![F_2]\!] e) \\ [\![\lambda v : T.F]\!]\, e & = & \lambda d.[\![F]\!]\, (e[v := d]) \end{array}$$

Semantics

Figure 4: Church-style Simply Typed Lambda Calculus

3.2 De Bruijn Notation

An alternative to referring to each bound variable by its name is to refer to each variable by its number, with number 1 denoting the most recently bound variable. This is the idea behind de Bruijn indices for lambda calculus [22, 4]. Figure 5 presents the syntax and the semantics of lambda calculus notation with de Bruijn indices. The environment maps the keyword *stack* to a stack (i.e., a list) of elements of the domain. If h is an element and l a list, then the notation $h : l$ denotes the list with the head h and the tail l. The abstraction pushes a value onto the stack; the index $\langle k \rangle$ retrieves the k-th element from the top of the stack.

3.3 Predicate Logic in Lambda Calculus

We next encode first-order logic with equality in lambda calculus. We use EQ to denote the binary equality relation. We assume that the interpretation of relation symbols is specified in the environment e. We introduce conjunction and negation as logical operations acting on booleans (the remaining propositional operations are defined in terms of $\wedge, \neg$, as usual). We use the abstraction in lambda calculus to encode bound variables of predicate calculus. This is the usual higher-order logic encoding of classical first-order logic, as used, for example, in Isabelle interactive theorem prover [58]. Figure 6 presents this encoding of quantifiers.

$$
\begin{array}{rcll}
\mathsf{Form} & = & \langle \mathsf{Nat} \rangle & \text{variable lookup} \\
 & & & \mathsf{Nat} = \{1,2,\ldots\} \\
 & | & \mathsf{Form}\ \mathsf{Form} & \text{function application} \\
 & | & \lambda : \mathsf{Type}.\mathsf{Form} & \text{function abstraction}
\end{array}
$$

Syntax

$$
\begin{array}{rcl}
[\![\langle i \rangle]\!]\, e & = & \mathsf{get}\, i\, e \\
[\![F_1\ F_2]\!]\, e & = & ([\![F_1]\!] e)\ ([\![F_2]\!] e) \\
[\![\lambda : T.F]\!]\, e & = & \lambda d.\ [\![F]\!]\, (\mathsf{push}\, d\, e)
\end{array}
$$

Semantics

$$
\begin{array}{rcl}
\mathsf{get}\, i\, e & = & \mathsf{nth}\, i\ (e\ \mathsf{stack}) \\
\mathsf{push}\, d\, e & = & e[\mathsf{stack} := d : (e\, \mathsf{stack})] \\
\mathsf{nth}\, 1\, (h : l) & = & h \\
\mathsf{nth}\, (i+1)\, (h : l) & = & \mathsf{nth}\, i\, l
\end{array}
$$

Auxiliary Functions

Figure 5: De Bruijn Form of Simply Typed Lambda Calculus

$$
\begin{array}{rcl}
\mathsf{EQ} & :: & \mathsf{rel}^2 \\
[\![\mathsf{EQ}]\!]\, x\, y & = & (x = y) \\
\wedge & :: & \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \\
[\![\wedge]\!]\, p\, q & = & p \wedge q \\
\neg & :: & \mathsf{bool} \to \mathsf{bool} \\
[\![\neg]\!]\, p & = & \neg p \\
\exists & :: & \mathsf{rel}^1 \to \mathsf{bool} \\
[\![\exists]\!]\, f & = & \exists o \in [\![\mathsf{obj}]\!].\ f\, o \\
\\
\exists v.F & \equiv & \exists(\lambda v : \mathsf{obj}.\ F) \\
\forall v.F & \equiv & \neg \exists v. \neg F
\end{array}
$$

Figure 6: First-Order Logic in Lambda Calculus

$$
\begin{array}{rcl}
\{F\} & \equiv & \forall(\lambda : \mathsf{obj}.F) \\
[F] & \equiv & \neg\{\neg F\}
\end{array}
$$

Quantifier Brackets

$$
\begin{array}{lll}
\text{When } \Gamma(r) = \mathsf{rel}^k & \text{then write} & r \\
 & \text{instead of} & r\langle k\rangle\langle k-1\rangle \ldots \langle 1\rangle
\end{array}
$$

Default Argument Rule

$$
\begin{array}{rcl}
\sim F & \equiv & (\lambda\lambda F)\langle 1\rangle\langle 2\rangle \\
F' & \equiv & (\lambda\lambda F)\langle 2\rangle\langle 2\rangle \\
\mathsf{card}^{\geq k} F & \equiv & \{^k\ (\lambda F)\langle 1\rangle \wedge \ldots \wedge (\lambda F)\langle k\rangle \wedge \\
 & & \quad \bigwedge_{1\leq i<j\leq k} \neg \mathsf{EQ}\langle i\rangle\langle j\rangle\ \}^k \\
\mathsf{card}^{=k} F & \equiv & \mathsf{card}^{\geq k} F \wedge \neg \mathsf{card}^{\geq k+1} F \\
(\sum_{i=1}^n \mathsf{Card}\, F_i) \geq k & \equiv & \bigvee_{\sum_{i=1}^n k_i = k} \bigwedge_{i=1}^n \mathsf{card}^{\geq k_i} F_i \\
(\sum_{i=1}^n \mathsf{Card}\, F_i) = k & \equiv & \bigvee_{\sum_{i=1}^n k_i = k} \bigwedge_{i=1}^n \mathsf{card}^{= k_i} F_i \\
\mathsf{disjoint}\, F_1, \ldots, F_n & \equiv & [\bigwedge_{1\leq i<j\leq n} \neg(F_i \wedge F_j)] \\
\mathsf{partition}\, F; F_1, \ldots, F_n & \equiv & \mathsf{disjoint}\, F_1, \ldots, F_n\ \wedge \\
 & & [F \Leftrightarrow \bigvee_{i=1}^n F_i] \\
F_1 \setminus F_2 & \equiv & F_1 \wedge \neg F_2
\end{array}
$$

Shorthands

Figure 7: de Bruijn form of Predicate Calculus

To remain within first-order logic, we require the quantifier $\exists$ to have monomorphic type $(\mathsf{obj} \to \mathsf{bool}) \to \mathsf{bool}$ (see also Section 3.7).

3.4 Implicit De Bruijn Indices

Figure 7 shows how we combine the encoding of first-order logic in higher-order logic and de Bruijn's notation for lambda calculus.

Example 1 First-order predicate calculus formula

$$\forall x \forall y.\ f(x,y) \Rightarrow A(x) \wedge B(y)$$

can be written in this notation as

$$[[f\langle 2\rangle\langle 1\rangle \Rightarrow A\langle 2\rangle \wedge B\langle 1\rangle]]$$

The outermost [ ] bracket acts as the quantifier $\forall x$; the variable x is referred to inside the formula as $\langle 2\rangle$ because it is the second innermost bound variable. The innermost [ ] bracket acts as $\forall y$; the variable y is referred to as $\langle 1\rangle$.

♦

The interpretation environment e contains both the stack for de Bruijn indices and the bindings of relation symbols such as A and f in Example 1. Relation symbols of predicate logic correspond to variables of type rel^k. We use the abstraction over de Bruijn indices $\lambda{:}T.F$ only when $T \equiv \mathsf{obj}$, and write this abstraction simply λF. For every environment e, the value $(e\,\mathsf{stack})$ is a list of elements of type obj.

We next introduce the Default Argument Rule: we omit de Bruijn indices from the expression $r\langle k\rangle\langle k{-}1\rangle \ldots \langle 1\rangle$ when r is a relation symbol, that is, when $\Gamma(r) = \mathsf{rel}^k$. We interpret every occurrence of variable r when $\Gamma(r) = \mathsf{rel}^k$ as $r\langle k\rangle\langle k{-}1\rangle \ldots \langle 1\rangle$.

Example 2 The Default Argument Rule means that instead of

$$[[f\langle 2\rangle\langle 1\rangle \Rightarrow A\langle 2\rangle \wedge B\langle 1\rangle]]$$

we write

$$[[f \Rightarrow (\lambda A)\langle 2\rangle \wedge B]]$$

when $\Gamma(f) = \mathsf{rel}^2$ and $\Gamma(A) = \Gamma(B) = \mathsf{rel}^1$.

♦

We lose no expressive power by the Default Argument Rule. For example, if we wish to denote $r\langle i_3\rangle\langle i_2\rangle\langle i_1\rangle$, we write $(\lambda\lambda\lambda r)\langle i_3\rangle\langle i_2\rangle\langle i_1\rangle$. Note that the Default Argument Rule applies only to the relation symbols, not to all subformulas, so $(\lambda\lambda\lambda r)$ with Default Argument rule is equivalent to r without Default Argument Rule. In general, if r is an n-ary relation, we write $((\lambda)^k r)\langle i_k\rangle\langle i_{k-1}\rangle \ldots \langle i_1\rangle$ where we would previously write $r\langle i_k\rangle\langle i_{k_1}\rangle \ldots \langle i_1\rangle$.

3.5 Shorthands

Figure 7 introduces some shorthands. Tilde $\sim$ swaps two topmost stack elements $\langle 1\rangle$ and $\langle 2\rangle$. Prime $'$ replaces the top $\langle 1\rangle$ with the element $\langle 2\rangle$. An expression $\mathsf{card}^{\geq k}F$, for an integer $k \geq 0$, corresponds to a counting quantifier in first-order logic [30]. A counting quantifier states that the number of elements with some property is greater than or equal to k. Figure 7 also introduces the shorthand for $\mathsf{card}^{=k}F$ and the shorthand Card for specifying a constraint on a sum of cardinalities. The shorthands containing $\leq$ are defined similarly.

These shorthands play two purposes. On the one hand they allow expressing certain properties in a more concise way. On the other hand, if we use the shorthands but give up the ability to refer to indices explicitly, we obtain a fragment of first-order logic that is equivalent to two-variable first-order logic with counting (Section 4) and therefore decidable [30].

Example 3 Using the shorthands, we write the formula

$$\forall x \forall y.\ f(x,y) \Rightarrow A(x) \wedge B(y)$$

as

$$[[f \Rightarrow A' \wedge B]]$$

The convenience of role logic is even more evident in larger formulas like

$$\begin{gathered}\forall x.\ A(x) \Rightarrow (\forall y. f(x,y) \Rightarrow B(y) \vee C(y)) \wedge \\ (\forall z. g(x,z) \Rightarrow D(z))\end{gathered}$$

which can be written as

$$[A \Rightarrow [f \Rightarrow B \vee C] \wedge [g \Rightarrow D]] \qquad (1)$$

$$\begin{array}{rcl}
F* & \equiv & \mathsf{rtrancl}\,(\lambda\lambda F)\,\langle 2\rangle\,\langle 1\rangle \\
[\![\mathsf{rtrancl}]\!]\, r\, x\, y & = & \exists n \geq 0. \exists z_0, \ldots, z_n.\ z_0 = x \wedge z_n = y \wedge \\
 & & \bigwedge_{i=0}^{n-1} r\, z_i\, z_{i+1} \\
F_1 \circ F_2 & \equiv & \{(\lambda\lambda F_1)\langle 3\rangle\langle 1\rangle \ \wedge\ (\lambda\lambda F_1)\langle 2\rangle\langle 1\rangle\} \\
F+ & \equiv & F \circ F* \\
\mathsf{acyclic}\ F & \equiv & \neg\{F+ \wedge \mathsf{EQ}\} \\
\mathsf{tree}\ F_1, \ldots, F_n & \equiv & \mathsf{acyclic}\ \bigvee_{i=1}^{n} F_i \\
 & \wedge & [(\bigvee_{i=1}^{n} F_i)* \Rightarrow \\
 & & \sum_{i=1}^{n} \mathsf{Card}\,(\sim F_i) \leq 1]
\end{array}$$

Figure 8: Transitive Closure Construct and Shorthands

Formulas of form (1) are useful for describing properties of first order structures that arise in shape analysis, see e.g. [48, 47, 71].

♦

For additional expressive power we introduce the reflexive-transitive closure operator $*$, with the semantics in Figure 8. We also introduce a shorthand for relation composition. The relation composition shorthand works when F_1 and F_2 both denote binary relations, when the resulting expression can be thought of as denoting a binary relation, as well as when F_1 denotes a set and F_2 denotes a binary relation, when the resulting expression denotes the set which is the image of F_1 under F_2. For the case of relation we also introduce a simpler definition in Figure 13 whose advantage is that it uses only two implicit indices.

3.6 Role Logic

Figure 9 summarizes the syntax of role logic. The semantics of role logic follows from Section 3.

We next explain the purpose of lambda abstraction in our logic.

3.7 Lambda Calculus for Predicate Definitions

In the resulting role logic of Figure 9 we retain the named variables in the environment, and we allow abstraction over those named variables. As a result, there two kinds of lambda abstraction: abstraction over de Bruijn indices and abstraction over named variables. Abstraction over a de Bruijn index is always over $\langle 1\rangle$ which denotes an object of type obj, such abstraction is written λF. The abstraction over a named variable may abstract over variables of more complex types and is written $\lambda x : T.F$. There is only one kind of lambda calculus application; both $(\lambda F_1)F_2$ and $(\lambda x : T.F_1)F_2$ are redexes.

The purpose of the named lambda abstraction $\lambda x : T.F$ is twofold. First, when $T \equiv \mathsf{obj}$, then we can write $\exists(\lambda x : \mathsf{obj}.F)$ as $\exists x.F$ as in the usual first-order predicate calculus. Second, when T is not obj, we can encode acyclic definitions of higher-order predicates that can be subsequently substituted away. Define the expression

$$\mathsf{let}\ P : T = F_1\ \mathsf{in}\ F_2$$

Form	$=$	Vars	named object or predicate
	$\mid$	$\langle\mathsf{Nat}\rangle$	de Bruijn index of an object variable
	$\mid$	EQ	equality between $\langle 1\rangle$ and $\langle 2\rangle$
	$\mid$	$\mathsf{Form} \land \mathsf{Form}$	conjunction
	$\mid$	$\neg\mathsf{Form}$	negation
	$\mid$	$\exists\mathsf{Form}$	existential quantification over objects
	$\mid$	$\lambda\mathsf{Form}$	de Bruijn abstraction over objects
	$\mid$	$\lambda\mathsf{Vars} : \mathsf{Type}\ .\ \mathsf{Form}$	abstraction over named variables
	$\mid$	$\mathsf{Form}\ \mathsf{Form}$	function application
	$\mid$	Form'	let $\langle 1\rangle$ be $\langle 2\rangle$ in F
	$\mid$	$\sim\mathsf{Form}$	relation inverse
	$\mid$	$\mathsf{card}^{\geq k}\mathsf{Form}$	at least k objects satisfy F
	$\mid$	$\mathsf{Form}*$	reflexive transitive closure

Figure 9: The Syntax of Role Logic

to be equivalent to

$$(\lambda P : T\ .\ F_2)F_1$$

Such definitions are very useful for describing complex data structures.

Note that acyclic definitions introduced through typed lambda calculus via bindings $\lambda x : T.F$ for $T \not\equiv \mathsf{bool}$ do not make the logic higher-order, because we define the the quantifier $\exists$ to always have the monomorphic type $(\mathsf{obj} \to \mathsf{bool}) \to \mathsf{bool}$, and the reflexive-transitive closure operator $*$ to have the type

$$(\mathsf{obj} \to \mathsf{obj} \to \mathsf{bool}) \to (\mathsf{obj} \to \mathsf{obj} \to \mathsf{bool})$$

Consider a well-typed formula F whose only free variables are relation symbols, and whose de Bruijn indices only refer to indices bound in the formula. Assume that we have applied the Default Argument Rule, so that all de Bruijn indices are explicit. Then we may treat de Bruijn abstraction as the usual abstraction over a disjoint set of variables. By strong normalization of simply typed lambda calculus [5], let F^0 be the normal form of F. We claim that in F^0 the only occurrence of lambda abstraction is within expressions of the form $\exists(\lambda x : \mathsf{obj}.F)$ or $\mathsf{rtrancl}(\lambda x : \mathsf{obj}.\lambda y : \mathsf{obj}.F)$.

To show the claim, consider an occurrence of $\lambda x : \mathsf{obj}.F_0$ in F^0. Let F_1 be the largest enclosing occurrence $\lambda x_1 : T_1. \ldots .\lambda x_n : T_n.\lambda x : \mathsf{obj}.F_0$. Then F_1 cannot be the entire F^0, because F^0 has type bool by subject reduction. F_1 cannot occur within some application $F_1 F_2$, because $F_1 F_2$ would constitute a redex and F^0 is in normal form. Hence, F_1 can only occur in an expression of the form $F_3 F_1$. Let us consider the "spine" [38] of $F_3 F_1$, so $F_3 \equiv F_n F_{n-1} \ldots F_4$ $n \geq 3$ and F_n is not an application. F_n is not an abstraction, because F^0 is in normal form. Hence, F_n can only be a variable or a constant.

The only variables or or constants that can, by the typing rules, be applied to an abstraction F_1 are $\exists$ and $\mathsf{rtrancl}$, so either $F_n \equiv \exists$ or $F_n \equiv \mathsf{rtrancl}$.

Consider the case $F_n \equiv \exists$. By the type of $\exists$, we conclude $F_3 \equiv F_n$ and $F_1 \equiv \lambda x : \mathsf{obj}.F_0$, as desired.

Consider the case $F_n \equiv \mathsf{rtrancl}$. Then $F_3 \equiv F_n$, and $F_1 \equiv \lambda u : \mathsf{obj}.\lambda v : \mathsf{obj}.G$, so either $u \equiv x$ and $F_1 \equiv \lambda x.\mathsf{obj}.F_0$ where $F_0 = \lambda v : \mathsf{obj}.G$, or $v \equiv x$ and $F_1 \equiv \lambda u : \mathsf{obj}.\lambda x : \mathsf{obj}.F_0$. This finishes the proof of the claim.

We conclude that typed lambda calculus allows us to use flexible definitions of higher-order predicates to structure our specifications while keeping the language first-order, because we may substitute away all definitions using strong normalization of the typed lambda calculus.

4 Role Logic Subset RL^2 and its Decidability

In this section we introduce a subset RL^2 of role logic (Figure 11) and show its decidability.

To show the decidability of RL^2, we give translations of formulas between the following four logics:

1. D^2: the formulas of the first-order logic with counting in which every subformula has at most two free variables (different subformulas may have different free variables);
2. C^2: the formulas of the two-variable logic with counting, which uses x and y as the only variable names; the satisfiability and finite satisfiability problem for C^2 was shown to be decidable in [30]; the satisfiability problem for C^2 was shown $\mathsf{NEXPTIME}$-complete in [57];
3. I^2: de Bruijn version of the two-variable logic with counting, which uses only de Bruijn indices $\langle 1\rangle$ and $\langle 2\rangle$;
4. RL^2: a subset of role logic that contains no explicit de Bruijn indices.

Figure 10 sketches the idea of the proof of equivalence of these four logics. We give translations of formulas from D^2 to C^2 (Section 4.2, Figure 15) from C^2 to I^2 (Section 4.3, Figure 18), from I^2 to RL^2 (Section 4.3, Figure 19) and from RL^2 to D^2 (Section 4.4, Figure 20). These translations imply that the satisfiability problem for these four logics are equivalent, so by decidability of C^2 [30] we conclude that all these logics are decidable.

$$\begin{tikzcd}
D^2 \arrow[r, "\text{Fig. 15}"] & C^2 \arrow[d, "\text{Fig. 18}"] \\
\mathsf{RL}^2 \arrow[u, "\text{Fig. 20}"] & I^2 \arrow[l, "\text{Fig. 19}"']
\end{tikzcd}$$

Figure 10: Showing Equivalence of Four Logics.

$$\begin{array}{lcll}
\mathsf{Form} & = & \mathsf{Vars} & \text{binary or unary relation symbol} \\
 & \mid & \mathsf{EQ} & \text{equality between } \langle 1 \rangle \text{ and } \langle 2 \rangle \\
 & \mid & \mathsf{Form} \wedge \mathsf{Form} & \text{conjunction} \\
 & \mid & \neg\mathsf{Form} & \text{negation} \\
 & \mid & \mathsf{Form}' & \text{let } \langle 1 \rangle \text{ be } \langle 2 \rangle \text{ in } F \\
 & \mid & \sim\!\mathsf{Form} & \text{relation inverse} \\
 & \mid & \mathsf{card}^{\geq k}\mathsf{Form} & \text{at least } k \text{ objects satisfy } F
\end{array}$$

Figure 11: The Syntax of RL^2 Subset of Role Logic

$$\begin{array}{rcl}
\mathsf{Nat}_2 & = & \{1, 2\} \\
e & :: & \mathsf{Nat}_2 \to \mathsf{obj} \\
[\![A]\!]e & = & [\![A]\!](e\,1) \\
[\![f]\!]e & = & [\![f]\!](e\,2, e\,1) \\
[\![\mathsf{EQ}]\!]e & = & (e\,2) = (e\,1) \\
[\![F_1 \wedge F_2]\!]e & = & ([\![F_1]\!]e) \wedge ([\![F_2]\!]e) \\
[\![\neg F]\!]e & = & \neg([\![F]\!]e) \\
[\![F']\!]e & = & [\![F]\!](e[1 \mapsto (e\,2)]) \\
[\![\sim\!F]\!]e & = & [\![F]\!](e[1 \mapsto (e\,2), 2 \mapsto (e\,1)]) \\
[\![\mathsf{card}^{\geq k} F]\!]e & = & |\{o \mid [\![F]\!](e[1 \mapsto o, 2 \mapsto (e\,1)])\}| \geq k
\end{array}$$

Figure 12: The Semantics of RL^2

quantifiers:

$$\begin{array}{rcl}
\{F\} & = & \mathsf{card}^{\geq 1} F \\
[F] & = & \neg\{\neg F\}
\end{array}$$

relation image:

$$F_A \,\text{`}\, F_r \;=\; \{F_A \wedge \sim\!F_r\}$$

weakest precondition:

$$\mathsf{wp}\, F_r\, F_A \;=\; [F_r \Rightarrow F_A]$$

Figure 13: Some Shorthands for RL^2

4.1 The Role Logic Subset RL^2

Figure 11 presents the two-variable role logic RL^2. Compared to the full role logic in Figure 9, RL^2 omits the constructs for creating definitions, the constructs for explicitly referring to object variables, and transitive closure. Figure 12 summarizes the semantics of RL^2; this semantics is in accordance with the semantics of the full role logic derived in Section 3. Figure 13 defines shorthands that illustrate some constructs definable in RL^2.

We show that RL^2 has precisely the same expressive power as the set of the formulas of logic C^2, which is shown decidable in [30] over the set of all models, as well as over the set of finite models.

4.2 Two-Variable Logics C^2 and D^2

Figure 14 presents the logic C^2 [30]. The logic C^2 is first-order logic with equality and counting, restricted to formulas that contain only two fixed variable names x and y.

In this section we argue that a more flexible restriction on variable names yields logic with same definable relations. Let $\mathsf{FV}(F)$ denote the free variables of formula F.

Definition 4 *A D^2 formula is a formula F of first-order logic with counting such that $|\mathsf{FV}(G)| \leq 2$ for every subformula G of F.*

Clearly every C^2 formula is a D^2 formula, but not vice versa, because the set of possible variables that may occur in D^2 formulas is countably infinite. The syntactic restriction on variables in Definition 4 is more general than in the definition in C^2, which makes D^2 more convenient for writing readable formulas.

We show that every D^2 formula is equivalent to a C^2 formula (modulo the renaming of free variables). Up to one technical detail, it suffices to rename bound variables in a D^2 formula to obtain a C^2 formula. We therefore derive the equivalence of D^2 and C^2 as a consequence of an observation about lambda calculus terms.

Definition 5 *Define the set of lambda calculus terms* $\mathsf{2VarTerms}$ *as the smallest set that satisfies the following conditions:*

1. *$v \in \mathsf{2VarTerms}$ if v is a variable and $c \in \mathsf{2VarTerms}$ if c is a constant;*
2. *if $T_1, T_2 \in \mathsf{2VarTerms}$ and $|\mathsf{FV}(T_1) \cup \mathsf{FV}(T_2)| \leq 2$, then $(T_1 T_2) \in \mathsf{2VarTerms}$;*

$$
\begin{array}{rcll}
\mathsf{Vars}_2 & = & \{x, y\} & \\
\mathsf{Form} & = & A(\mathsf{Vars}_2) & \text{atomic formula with unary relation } A \\
& | & f(\mathsf{Vars}_2, \mathsf{Vars}_2) & \text{atomic formula with binary relation } f \\
& | & \mathsf{Vars}_2 = \mathsf{Vars}_2 & \text{equality between objects} \\
& | & \mathsf{Form} \wedge \mathsf{Form} & \text{conjunction} \\
& | & \neg \mathsf{Form} & \text{negation} \\
& | & \exists^{\geq k} \mathsf{Vars}_2.\, \mathsf{Form} & \text{at least } k \text{ objects satisfy formula}
\end{array}
$$

Figure 14: The Syntax of Two-Variable Logic with Counting C^2

3. *if* $T \in \mathsf{2VarTerms}$*, v is a variable, and* $|\mathsf{FV}(T) \cup \{v\}| \leq 2$*, then* $\lambda v.T \in \mathsf{2VarTerms}$.

From Definition 5 it follows that if $T \in \mathsf{2VarTerms}$, then $|\mathsf{FV}(T_1)| \leq 2$ for every subterm T_1 of T. Moreover, if $\lambda v.T \in \mathsf{2VarTerms}$ and $v \notin \mathsf{FV}(T)$, then $|\mathsf{FV}(T)| \leq 1$.

We next define the set $\mathsf{capt}(v, F)$ of those bound variables z in formula F such that v occurs in the scope of a binding of z.

Definition 6

$$
\begin{array}{rcl}
\mathsf{capt}(v, u) & = & \emptyset, \ \textit{if } u \textit{ is a variable} \\
\mathsf{capt}(v, F_1 F_2) & = & \mathsf{capt}(v, F_1) \cup \mathsf{capt}(v, F_2) \\
\mathsf{capt}(v, \lambda u.F) & = & \begin{cases} \mathsf{capt}(v, F) \cup \{u\}, & \textit{if } v \in \mathsf{FV}(\lambda u.F) \\ \emptyset, & \textit{otherwise} \end{cases}
\end{array}
$$

As usual, we say that T and T' are α-equivalent if T' can be obtained from T by renaming bound variables.

Lemma 7 *For every* $T \in \mathsf{2VarTerms}$ *with* $\mathsf{FV}(T) \subseteq \{u, v\}$ *there exists a term* $T' = \mathsf{norm}(T)$ *such that T' is α-equivalent to T, all bound variables in T' are among $\{x, y\}$, and either*

1. $\mathsf{capt}(u, T') \subseteq \{x\}$ *and* $\mathsf{capt}(v, T') \subseteq \{y\}$*, or*
2. $\mathsf{capt}(u, T') \subseteq \{y\}$ *and* $\mathsf{capt}(v, T') \subseteq \{x\}$.

Proof. Let $\mathsf{FV}(T) \subseteq \{u, v\}$. Without loss of generality we may assume that $\{u, v\} \cap \{x, y\} = \emptyset$. The proof is by induction on the structure of terms.

1. $T = u$ for a variable u. Let $T = T'$, clearly $\mathsf{capt}(u, T') = \mathsf{capt}(v, T') = \emptyset$.
2. $T = T_1 T_2$. Let $T_1' = \mathsf{norm}(T_1)$ and $T_2' = \mathsf{norm}(T_2)$ by induction hypothesis. Assume $\mathsf{capt}(u, T_1') \subseteq \{x\}$ and $\mathsf{capt}(v, T_1') \subseteq \{y\}$ (the other case is symmetric). We consider two cases for T_2'.
   (a) $\mathsf{capt}(u, T_2') \subseteq \{x\}$ and $\mathsf{capt}(v, T_2') \subseteq \{y\}$. Then let $\mathsf{norm}(T) = T_1' T_2'$.
   (b) $\mathsf{capt}(u, T_2') \subseteq \{y\}$ and $\mathsf{capt}(v, T_2') \subseteq \{x\}$. Let T_2'' be the result of swapping in T_2' all occurrences of bound variables x and y. Then $\mathsf{capt}(u, T_2'') \subseteq \{x\}$ and $\mathsf{capt}(v, T_2'') \subseteq \{y\}$, so we let $\mathsf{norm}(T) = T_1' T_2''$.

   In both cases, $\mathsf{capt}(u, \mathsf{norm}(T)) \subseteq \{x\}$ and $\mathsf{capt}(v, \mathsf{norm}(T)) \subseteq \{y\}$.
3. $T = \lambda w.T_1$. $|\{u, v\}| = 2$ and $|\mathsf{FV}(T_1) \cup \{w\}| \leq 2$ by the definition of $\mathsf{2VarTerms}$, so it cannot be the case that both $u \in \mathsf{FV}(T_1)$ and $v \in \mathsf{FV}(T_1)$. Since $\mathsf{FV}(T_1) \subseteq \{u, v, w\}$, we conclude that $\mathsf{FV}(T_1) \subseteq \{u, w\}$ or $\mathsf{FV}(T_1) \subseteq \{v, w\}$.

   Suppose therefore that $\mathsf{FV}(T_1) \subseteq \{u, w\}$ (the case $\mathsf{FV}(T_1) \subseteq \{v, w\}$ is symmetric). By induction hypothesis, let $T_1' = \mathsf{norm}(T_1)$. Assume $\mathsf{capt}(u, T_1) \subseteq \{x\}$ and $\mathsf{capt}(w, T_1) \subseteq \{y\}$ (the case $\mathsf{capt}(u, T_1) \subseteq \{y\}$ and $\mathsf{capt}(w, T_1) \subseteq \{x\}$ is symmetric). Let $\mathsf{norm}(T) = \lambda x.(F_1[w := x])$. Then $\mathsf{capt}(u, \mathsf{norm}(T)) \subseteq \{x\}$ and $\mathsf{capt}(v, \mathsf{norm}(T)) = \emptyset \subseteq \{y\}$.

■

To apply Lemma 7 to D^2 formulas, we represent all logical operations and quantifiers as constants. Variables in a lambda term then correspond to first-order variables. To ensure that the representation of formulas satisfies the condition $|\mathsf{FV}(T) \cup \{v\}| \leq 2$ for each term $\lambda v.T$, we require the following condition:

$$
\begin{array}{c}
\text{For every formula } \exists^{\geq k} x.\, F, \\
\text{either } x \in \mathsf{FV}(F) \text{ or } F \equiv \mathsf{true}.
\end{array} \tag{2}
$$

We ensure this condition by applying the rule

$$
\exists^{\geq k} x.\, F \;\cong\; F \wedge \exists^{\geq k} x.\, \mathsf{true}
$$

for $x \notin \mathsf{FV}(F)$.

After ensuring the condition (2), we apply the translation in Figure 15. Lemma 7 justifies the correctness of the translation. The translated formula is of the same size as the original formula. The translation can clearly be performed in polynomial time, including the process of ensuring the condition (2). The translation time can be made close to linear by delaying the application of the substitution $[w := x]$ and the swap operation.

4.3 From C^2 to RL^2 via I^2

In this section we introduce logic I^2 (Figure 16). We then give translations from C^2 to I^2 (Figure 18), and from I^2 to RL^2 (Figure 19).

$$
\begin{array}{rcl}
\mathcal{T}_{DC}[\![A(v)]\!] & = & A(v) \\
\mathcal{T}_{DC}[\![f(u,v)]\!] & = & f(u,v) \\
\mathcal{T}_{DC}[\![\neg F]\!] & = & \neg \mathcal{T}_{DC}[\![F]\!] \\
\mathcal{T}_{DC}[\![F_1 \wedge F_2]\!] & = & \begin{cases} F_1' \wedge F_2', & \text{if } \mathsf{capt}(u, F_1'), \mathsf{capt}(u, F_2') \subseteq \{x\} \\ & \quad \mathsf{capt}(v, F_1'), \mathsf{capt}(v, F_2') \subseteq \{y\} \\ & \quad \text{or} \\ & \quad \mathsf{capt}(u, F_1'), \mathsf{capt}(u, F_2') \subseteq \{y\} \\ & \quad \mathsf{capt}(v, F_1'), \mathsf{capt}(v, F_2') \subseteq \{x\} \\ F_1' \wedge (\mathsf{swap}\ F_2'), & \text{otherwise} \end{cases} \\
& & \mathsf{FV}(F_1 \wedge F_2) = \{u, v\} \\
& & F_1' = \mathcal{T}_{DC}[\![F_1]\!] \\
& & F_2' = \mathcal{T}_{DC}[\![F_2]\!] \\
& & \begin{array}{lcl} \mathsf{swap}\,(A(v)) & = & A(s\,u, s\,v) \\ \mathsf{swap}\,(f(u,v)) & = & f(s\,u, s\,v) \\ \mathsf{swap}\,(\neg F) & = & \neg(\mathsf{swap}\, F) \\ \mathsf{swap}\,(F_1 \wedge F_2) & = & \mathsf{swap}\, F_1 \ \wedge \ \mathsf{swap}\, F_2 \\ \mathsf{swap}\,(\exists^{\geq k} v.\, F) & = & \exists^{\geq k} (s\,v).\, (\mathsf{swap}\, F) \end{array} \\
& & \quad s\,x = y, \quad s\,y = x \\
& & \quad s\,u = u, \ \text{if } u \notin \{x, y\} \\
\mathcal{T}_{DC}[\![\exists^{\geq k} w.\, F]\!] & = & \begin{cases} \exists^{\geq k} x.\, (F'[w := x]), & \text{if } \mathsf{capt}(u, F') \subseteq \{x\}, \mathsf{capt}(w, F') \subseteq \{y\} \\ \exists^{\geq k} y.\, (F'[w := y]), & \text{if } \mathsf{capt}(u, F') \subseteq \{y\}, \mathsf{capt}(w, F') \subseteq \{x\} \end{cases} \\
& & \mathsf{FV}(F) \subseteq \{u, w\} \\
& & F' = \mathcal{T}_{DC}[\![F]\!]
\end{array}
$$

Figure 15: Translation of D^2 formulas to C^2 formulas.

$$
\begin{array}{rcll}
\mathsf{Form} & = & A(\langle \mathsf{Nat}_2 \rangle) & \text{atomic formula with unary relation } A \\
& | & f(\langle \mathsf{Nat}_2 \rangle, \langle \mathsf{Nat}_2 \rangle) & \text{atomic formula with binary relation } f \\
& | & \langle \mathsf{Nat}_2 \rangle = \langle \mathsf{Vars}_2 \rangle & \text{equality between objects} \\
& | & \mathsf{Form} \wedge \mathsf{Form} & \text{conjunction} \\
& | & \neg \mathsf{Form} & \text{negation} \\
& | & \mathsf{card}^{\geq k} \mathsf{Form} & \text{at least } k \text{ objects satisfy formula}
\end{array}
$$

Figure 16: The Syntax of Intermediate Logic I^2

$$
\begin{array}{rcl}
e & :: & \mathsf{Nat}_2 \to \mathsf{Vars}_2 \\
\mathcal{T}_{IC}\llbracket A(\langle i\rangle)\rrbracket e & = & A(e\,i) \\
\mathcal{T}_{IC}\llbracket f(\langle i_1\rangle, \langle i_2\rangle)\rrbracket e & = & f(e\,i_1, e\,i_2) \\
\mathcal{T}_{IC}\llbracket \langle i_1\rangle{=}\langle i_2\rangle\rrbracket e & = & (e\,i_1) = (e\,i_2) \\
\mathcal{T}_{IC}\llbracket F_1 \wedge F_2\rrbracket e & = & (\mathcal{T}_{IC}\llbracket F_1\rrbracket e) \wedge (\mathcal{T}_{IC}\llbracket F_2\rrbracket e) \\
\mathcal{T}_{IC}\llbracket \neg F\rrbracket e & = & \neg(\mathcal{T}_{IC}\llbracket F\rrbracket e) \\
\mathcal{T}_{IC}\llbracket \mathsf{card}^{\geq k} F\rrbracket e & = & \exists^{\geq k} v.\, (\mathcal{T}_{IC}\llbracket F\rrbracket [1 \mapsto v, 2 \mapsto (e\,1)]) \\
 & & v = s(e\,1) \\
 & & s\,x = y, \quad s\,y = x
\end{array}
$$

correctness criterion:

$$\llbracket \mathcal{T}_{IC}\llbracket F\rrbracket e\rrbracket e_C = \llbracket F\rrbracket(e_C \circ e)$$

Figure 17: Translating I^2 formulas to C^2 formulas

Intermediate logic. Figure 16 presents logic I^2. I^2 is a version of C^2 that uses two de Bruijn indices instead of variables. We introduce I^2 to separate the the translation of C^2 formulas to RL^2 in two phases: the first phase introduces de Bruijn indices, and the second phase introduces Default Argument Rule.

For the sake of illustration, we first present a converse translation, from I^2 to C^2, although we do not need this translation to show the equivalence of D^2, C^2, I^2, and RL^2.

From I^2 to C^2. Figure 17 presents the translation of I^2 into C^2. This translation amounts to introducing alternatively variables x and y for each counting quantifier, and resolving the indices appropriately. Using the criterion in Figure 17, the correctness of the translation follows by induction on the structure of formulas.

From C^2 to I^2. We turn to the translation from C^2 to I^2. Consider the C^2 formula

$$F \;\equiv\; \exists^{\geq 1} y.\, (\exists^{\geq 1} x.\, (\exists^{\geq 1} x.\, P(x,y)) \wedge Q(x,y))$$

The subformula $P(x,y)$ of F refers to the variable y, which is the 3rd bound variable starting from the innermost one. Therefore, the straightforward replacement of variables by de Bruijn indices would require the access to $\langle 3\rangle$. To address this problem, the translation from C^2 to I^2 involves a preparatory "alternating transformation" on C^2 formulas. For every formula F, let $B(F)$ denote some purely propositional combination of F and perhaps some other formulas. The alternating transformation eliminates all subformulas of the form $\exists^{\geq k_1} v.\, B(\exists^{\geq k_2} v.\, G(v))$ for $v \in \mathsf{Vars}_2$. In the resulting formula, the sequence of bound variables along any path in the formula tree is alternating, that is, satisfies the regular expression $(y|\epsilon)(xy)^*(x|\epsilon)$.

For the purpose of alternating transformation, we add the disjunction $\vee$ to the language. We show how to eliminate successive quantification over x from $\exists^{\geq k_1} x.\, B(\exists^{\geq k_2} x.\, G)$ (the case of $\exists^{\geq k_1} y.\, B(\exists^{\geq k_2} y.\, G)$ is analogous). First, transform B into disjunction of canonical conjunctions of formulas H, where each H satisfies one of the following three conditions:

$$
\begin{array}{rcl}
e & :: & \mathsf{Vars}_2 \to \mathsf{Nat}_2 \\
\mathcal{T}_{CI}\llbracket A(v)\rrbracket e & = & A(\langle e\,v\rangle) \\
\mathcal{T}_{CI}\llbracket f(v_1, v_2)\rrbracket e & = & f(\langle e\,i_1\rangle, \langle e\,i_2\rangle) \\
\mathcal{T}_{CI}\llbracket v_1{=}v_2\rrbracket e & = & \langle e\,v_1\rangle = \langle e\,i_2\rangle \\
\mathcal{T}_{CI}\llbracket F_1 \wedge F_2\rrbracket e & = & (\mathcal{T}_{CI}\llbracket F_1\rrbracket e) \wedge (\mathcal{T}_{CI}\llbracket F_2\rrbracket e) \\
\mathcal{T}_{CI}\llbracket \neg F\rrbracket e & = & \neg(\mathcal{T}_{CI}\llbracket F\rrbracket e) \\
\mathcal{T}_{CI}\llbracket \exists^{\geq k} x.\, F\rrbracket e & = & \mathsf{card}^{\geq k}(\mathcal{T}_{CI}\llbracket F\rrbracket [x \mapsto 1, y \mapsto 2]) \\
 & & \text{invariant: } e\,y = 1 \\
\mathcal{T}_{CI}\llbracket \exists^{\geq k} y.\, F\rrbracket e & = & \mathsf{card}^{\geq k}(\mathcal{T}_{CI}\llbracket F\rrbracket [y \mapsto 1, x \mapsto 2]) \\
 & & \text{invariant: } e\,x = 1
\end{array}
$$

correctness criterion:

$$\llbracket \mathcal{T}_{CI}\llbracket F\rrbracket e\rrbracket e_I = \llbracket F\rrbracket(e_I \circ e)$$

Figure 18: Translating normalized C^2 formulas to I^2 formulas

$C1)$ H is quantifier-free;

$C2)$ H is of the form $\exists^{\geq k} v.\, G(v)$ for $v \in \mathsf{Vars}_2$;

$C2)$ H is of the form $\neg\exists^{\geq k} v.\, G(v)$ for $v \in \mathsf{Vars}_2$;

Let $B \equiv \bigvee_{i=1}^{n} B_i$ where each B_i is a canonical conjunction (cube) of formulas satisfying conditions $C1)$, $C2)$, $C3)$. Because $B_i \wedge B_j$ is contradictory for distinct cubes B_i and B_j, the sets of objects o satisfying different B_i are disjoint, so

$$|\{o \mid \llbracket B\rrbracket e[v \to o]\}| = \sum_{i=1}^{n} |\{o \mid \llbracket B_i\rrbracket e[v \to o]\}|$$

We can therefore replace counting quantifier on B with a propositional combination of counting quantifiers on B_i for $1 \leq i \leq n$ (as in quantifier elimination for boolean algebras, [67], [49, Section 3.2]). Specifically,

$$\exists^{\geq k_1} x.\, B \;\cong \bigvee_{\sum_{j=1}^{n} l_j = k_1} \; \bigwedge_{i=1}^{n} \exists^{\geq l_i} x.\, B_i \tag{3}$$

It is therefore sufficient to eliminate the successive quantification over x in $\exists^{\geq k_1} x.\, B_i(\exists^{\geq k_2} x.\, G)$. Group the conjuncts in B_i as follows. Let $\mathsf{FV}(F)$ denote free variables of formula F. Let $P(x)$ be the conjunction of conjuncts C of B_i such that $x \in \mathsf{FV}(C)$, and let Q be the conjunction of all conjuncts C of B_i such that $x \notin \mathsf{FV}(C)$. All occurrences of $\exists^{\geq k_2} x.\, G$ in B_i are in Q. We have

$$\exists^{\geq k_1} x.\, B_i \;\cong\; \exists^{\geq k_1} x.\, Q \wedge P(x) \;\cong\; Q \wedge \exists^{\geq k_1} x.\, P(x)$$

where the last equivalence follows easily by definition of the counting quantifier $\exists^{\geq x} k_1$. In the resulting formula $Q \wedge \exists^{\geq x} k_1.\, P(x)$, the subformula $\exists^{\geq k_2} x.\, G$ is in Q and is therefore not in the scope of the original quantifier. By repeating this transformation we ensure that all quantifiers are alternating.

$$
\begin{array}{rcl}
\mathcal{T}_{IR}[\![A(\langle 1\rangle)]\!] & = & A \\
\mathcal{T}_{IR}[\![A(\langle 2\rangle)]\!] & = & A' \\
\mathcal{T}_{IR}[\![f(\langle 2\rangle,\langle 1\rangle)]\!] & = & f \\
\mathcal{T}_{IR}[\![f(\langle 1\rangle,\langle 2\rangle)]\!] & = & \sim f \\
\mathcal{T}_{IR}[\![f(\langle 2\rangle,\langle 2\rangle)]\!] & = & f' \\
\mathcal{T}_{IR}[\![f(\langle 1\rangle,\langle 1\rangle)]\!] & = & \sim(f') \\
\mathcal{T}_{IR}[\![\langle 2\rangle = \langle 1\rangle]\!] & = & \mathsf{EQ} \\
\mathcal{T}_{IR}[\![\langle 1\rangle = \langle 2\rangle]\!] & = & \mathsf{EQ} \\
\mathcal{T}_{IR}[\![\langle 1\rangle = \langle 1\rangle]\!] & = & \mathsf{true} \\
\mathcal{T}_{IR}[\![\langle 2\rangle = \langle 2\rangle]\!] & = & \mathsf{true} \\
\mathcal{T}_{IR}[\![F_1 \wedge F_2]\!] & = & \mathcal{T}_{IR}[\![F_1]\!] \wedge \mathcal{T}_{IR}[\![F_2]\!] \\
\mathcal{T}_{IR}[\![\neg F]\!] & = & \neg\mathcal{T}_{IR}[\![F]\!] \\
\mathcal{T}_{IR}[\![\mathsf{card}^{\geq k} F]\!] & = & \mathsf{card}^{\geq k}\mathcal{T}_{IR}[\![F]\!]
\end{array}
$$

correctness criterion:

$$[\![\mathcal{T}_{IR}[\![F]\!]]\!]e_I = [\![F]\!]e_I$$

Figure 19: Translating I^2 formulas to RL^2 formulas

$$
\begin{array}{rcl}
e\,0 & \in & \mathsf{Nat} \\
e\,k & \in & \{y_1, y_2, \ldots\} \quad \text{for } k \in \{1,2\} \\
\mathcal{T}_{RD}[\![A]\!]e & = & A(e\,1) \\
\mathcal{T}_{RD}[\![f]\!]e & = & f(e\,2, e\,1) \\
\mathcal{T}_{RD}[\![\mathsf{EQ}]\!]e & = & (e\,2) = (e\,1) \\
\mathcal{T}_{RD}[\![F_1 \wedge F_2]\!]e & = & (\mathcal{T}_{RD}[\![F_1]\!]e) \wedge (\mathcal{T}_{RD}[\![F_2]\!]e) \\
\mathcal{T}_{RD}[\![\neg F]\!]e & = & \neg(\mathcal{T}_{RD}[\![F]\!]e) \\
\mathcal{T}_{RD}[\![\mathsf{card}^{\geq k} F]\!]e & = & \exists^{\geq k} v.\, [\![F]\!]e[0 \mapsto n, 1 \mapsto v, 2 \mapsto (e\,1)] \\
 & & v = y_n \\
 & & n = 1 + e\,0 \\
\mathcal{T}_{RD}[\![\sim F]\!]e & = & \mathcal{T}_{RD}[\![F]\!](e[1 \mapsto (e\,2), 2 \mapsto (e\,1)]) \\
\mathcal{T}_{RD}[\![F']\!]e & = & \mathcal{T}_{RD}[\![F]\!](e[1 \mapsto (e\,2)])
\end{array}
$$

correctness criterion:

$$[\![\mathcal{T}_{RD}[\![F]\!]e]\!]e_C = [\![F]\!](e_C \circ e)$$

result is in D^2:

$$\mathsf{FV}(\mathcal{T}_{RD}[\![F]\!]e) \subseteq \{e\,1, e\,2\}$$

Figure 20: Translating RL^2 formulas to D^2 formulas.

After the alternating transformation, the translation from C^2 to I^2 is straightforward, and is presented in Figure 18. The correctness of the translation follows by induction of the structure of formulas. The translation in Figure 18 runs in linear time and produces an I^2 formula whose size is linear in the size of the original C^2 formula.

The alternating transformation that precedes the translation may cause exponential blowup of the formula size due to translation to disjunctive normal form, but for most formulas the transformation need not be applied. Moreover, if we allow introducing new predicate names, then we may replace $\exists^{\geq k_1} x.\, B(\exists^{\geq k_2} x.\, G(x,y))$ with $\exists^{\geq k_1} x.\, B(P(y))$ and conjoin the topmost formula with the formula $\forall y. P(y) \iff \exists^{\geq k_2} x.\, G(x,y)$. Such transformation can be performed in linear time and preserves the satisfiability of formulas (see [30, Section 2.1, Page 18] and [30, Lemma 2.3]).

From I^2 to RL^2. Figure 19 presents the translation from I^2 to RL^2, which is simple and does not require a translation environment. The translation algorithm runs in linear time and produces a RL^2 formula whose size is linear in the size of the original I^2 formula.

4.4 From RL^2 to D^2: Closing the Loop

In the final step, we provide a translation from RL^2 formulas to D^2 formulas. The logic D^2 is a convenient target of translation of RL^2 formulas. (Namely, a simple attempt at translation from RL^2 to I^2 runs into the difficulty of the following form. Formula $(\mathsf{card}^{\geq 1} f)'$ is equivalent to $\mathsf{card}^{\geq 1} f(\langle 3\rangle, \langle 1\rangle)$ which uses index $\langle 3\rangle$ not available in I^2. Similarly, an attempt to translate from RL^2 to C^2 runs into difficulty of variable capture.)

Figure 20 presents the translation from RL^2 to D^2. The correctness of the translation follows by induction on the structure of formulas. Furthermore, each subformula G_1 of a formula $\mathcal{T}_{RD}[\![F]\!]e$ is of the form $G_1 \equiv \mathcal{T}_{RD}[\![G]\!]e_1$ for some G and r_1, and by induction it follows that the free variables of $\mathcal{T}_{RD}[\![G]\!]e_1$ are among $\{e_1\,1, e_1\,2\}$. Therefore, $|\mathsf{FV}(G_1)| \leq 2$ and the result of translation is a D^2 formula.

Summary As indicated in Figure 10, we have presented translations from D^2 to C^2, from C^2 to I^2, from I^2 to RL^2, and from RL^2 to D^2. We conclude that D^2, C^2, I^2, and RL^2 are all equivalent logics, and, by [30], decidable.

The satisfiability problem for C^2 formulas is shown to be NEXPTIME-complete in [57]. We have observed that there are efficient polynomial transformations of formulas from D^2 to C^2, from C^2 to I^2, from I^2 to RL^2 and from RL^2 to D^2 that yield formulas equivalent for satisfiability. (Moreover, all transformations except from C^2 to I^2 yield equivalent formulas in the same vocabulary.) As a result, the satisfiability problem of all these logics is NEXPTIME-complete.

5 Applications of Role Logic

We next present three applications of role logic. In Section 5.1 we present a shape analysis technique based on generating verification conditions in RL^2 and applying the decision procedure for RL^2. In Section 5.2 we note that boolean shape analysis constraints [48] are a subset of constraints expressible in role logic. In Section 5.3 we show that a different subset of RL^2 corresponds to an expressive description logic [1, Chapter 5].

5.1 Static Analysis Based on RL^2

This section shows how to use the decidability of RL^2 for static analysis of imperative programs. Figure 21 presents the syntax of a simple imperative language. Figure 22 presents predicates in RL^2 that describe the meaning of statements in this language.

Program state. The state of the program is a first-order structure interpreting the language $L = \mathcal{A} \cup \mathcal{F}$ where $\mathcal{A}$ is a finite set of unary predicates and $\mathcal{F}$ is a finite set of binary predicates. We fix a countable universe of objects obj, and assume that each structure has the same universe obj. To specify the structure, it suffices to give the set $eA \subseteq \mathsf{obj}$ for each unary predicate $A \in \mathcal{A}$, and a binary relation $ef \subseteq \mathsf{obj} \times \mathsf{obj}$ for each binary predicate $f \in \mathcal{F}$.

Extended language. For each $k \in \{\epsilon, 0, 1, \ldots\}$ we define the language $L_{(k)}$. We identify $L_{(\epsilon)}$ with L, $A_{(\epsilon)}$ with A and $f_{(\epsilon)}$ with f. For $k \in \{0, 1, \ldots\}$, we let $A_{(k)}$ be a fresh unary predicate symbol, and $f_{(k)}$ a fresh binary predicate symbol, and $L_{(k)}$ be the set of all $A_{(k)}$ and $f_{(k)}$. The notation $\mathsf{formRen}\,(i \to j)\,F$ for $i, j \in \{\epsilon, 0, 1, 2 \ldots\}$ denotes a formula resulting from F by replacing all elements of $L_{(i)}$ with the corresponding elements of $L_{(j)}$.

Describing relations in the extended language. The meaning of each statement in our imperative language is a binary relation on L-structures. We describe a binary relation on structures with an RL^2 formula in the language $L_{(0)} \cup L_{(\epsilon)}$. The predicates in $L_{(\epsilon)}$ denote the state components in the final state; the predicates in $L_{(0)}$ denote the state components in the initial state. If F is a formula in language $L_{(\epsilon)}$, then $\overline{F}$ is a shorthand for the formula $\mathsf{formRen}\,(\epsilon \to 0)\,F$ in the language $L_{(0)}$; the purpose of $\overline{F}$ is to denote the value of the formula F evaluated in the initial state.

Define the renaming operator $\mathsf{strucRen}\,(i \to j)$ such that if $e_{(i)}$ is an $L_{(i)}$-structure, then $e_{(j)} = \mathsf{strucRen}\,(i \to j)\,e_{(i)}$ is an $L_{(j)}$-structure such that $e_{(j)}\,A_{(j)} = e_{(i)}\,A_{(i)}$ and $e_{(j)}\,f_{(j)} = e_{(i)}\,f_{(i)}$ for all $A, f \in L$. Then the relation on L-structures denoted by an RL^2 formula F in language $L_{(0)} \cup L_{(\epsilon)}$ is $\{\langle e, e' \rangle \mid [\![F]\!]((\mathsf{strucRen}\,(\epsilon \to 0)\,e) \;\cup\; e')\}$.

Assignment statements. The imperative language in Figure 22 contains three forms of assignment statements.

The statement $A := F$ evaluates to the formula F, which denotes a unary predicate. The statement makes A true precisely for those object for which F was true in the initial state. Unary predicates other than A as well as binary predicates remain unchanged.

The statement $F_1.f := F_2$ generalizes the statement $x.f = y$ in a language like Java by allowing simultaneous modification of fields of a set of objects. Formula F_1 specifies the set of objects whose fields are modified. Formula F_2 specifies the new value of the field f for objects in F_1. Unary predicates and binary predicates other than f remain unchanged. Note that F_2 may specify a relation, which is particularly interesting when F_1 denotes a set with more then one element because it allows the value of the field to depend on the source object of the field. As a special case, $F_1.f := g$ copies the entire field g into field f for all objects in the set given by F_1, and, in particular, $\mathsf{true}.f := g$ copies the field g into f. The statement $F_1.{\sim}f := F_2$ is dual to $F_1.f := F_2$, and updates the inverse of the predicate f.

Statements for specification. The statement $\mathsf{assume}\,F$ filters out the state transitions for which F does not hold in the initial state. The statement $\mathsf{assert}\,F$ behaves arbitrarily if the condition given by F does not hold in the initial state. The state contains an additional predicate Error, which makes it easier to detect that an arbitrary behavior

$$
\begin{aligned}
[\![P_1 \Rightarrow P_2]\!] &= ([\![S_1]\!] \wedge \\
&\qquad \neg[\![S_2]\!](B_1 \mapsto A_1, \ldots, B_n \mapsto A_n)) \\
&\qquad \text{is not satisfiable, where:} \\
&\qquad P_1(A_1, \ldots, A_n) = S_1 \\
&\qquad P_2(B_1, \ldots, B_n) = S_2 \\
&\qquad [\![S_2]\!] \text{ has no fresh predicates} \\
[\![A := F]\!] &= [A \iff \overline{F}] \wedge \mathsf{modUnary}\,A \\
[\![F_1.f := F_2]\!] &= [\overline{F_1} \Rightarrow [f \iff \overline{F_2}]] \wedge \\
&\qquad [\neg\overline{F_1} \Rightarrow [f \iff \overline{f}]] \wedge \\
&\qquad \mathsf{modBinary}\ f \\
[\![F_1.{\sim}f := F_2]\!] &= [\overline{F_1} \Rightarrow [{\sim}f \iff \overline{F_2}]] \wedge \\
&\qquad [\neg\overline{F_1} \Rightarrow [{\sim}f \iff {\sim}\overline{f}]] \wedge \\
&\qquad \mathsf{modBinary}\ f \\
[\![P(F_1, \ldots, F_n)]\!] &= [\![S]\!](A_1 \mapsto \overline{F_1}, \ldots, A_n \mapsto \overline{F_n}) \\
&\qquad \text{where } P(A_1, \ldots, A_n) = S \\
[\![\mathsf{assume}\,F]\!] &= \overline{F} \wedge \mathsf{skip} \\
[\![\mathsf{assert}\,F]\!] &= \overline{F} \Rightarrow \mathsf{skip} \\
[\![\mathsf{spec}\,F]\!] &= [\![F]\!] \\
[\![s_1 \wedge s_2]\!] &= [\![s_1]\!] \wedge [\![s_2]\!] \\
[\![s_1 \vee s_2]\!] &= [\![s_1]\!] \vee [\![s_2]\!] \\
[\![s_1 ; s_2]\!] &= \mathsf{formRen}\,(\epsilon \to k)\,[\![s_1]\!] \wedge \\
&\qquad ([\neg\mathsf{Error}] \Rightarrow \mathsf{formRen}\,(0 \to k)\,[\![s_2]\!]) \\
&\qquad k - \text{fresh element of } \{1, 2, \ldots\} \\
[\![\mathsf{modify}\ E]\!] &= \mathcal{M}[\![E]\!] \\
\mathsf{modUnary}\,A &\equiv \textstyle\bigwedge_{B \neq A}[B \iff \overline{B}] \wedge \\
&\qquad \textstyle\bigwedge_g [[g \iff \overline{g}]] \wedge \\
&\qquad [\mathsf{Error} \iff \overline{\mathsf{Error}}] \\
\mathsf{modBinary} f &\equiv \textstyle\bigwedge_B [B \iff \overline{B}] \wedge \\
&\qquad \textstyle\bigwedge_{g \neq f} [[g \iff \overline{g}]] \wedge \\
&\qquad [\mathsf{Error} \iff \overline{\mathsf{Error}}] \\
\mathsf{skip} &\equiv \textstyle\bigwedge_B [B \iff \overline{B}] \wedge \\
&\qquad \textstyle\bigwedge_g [[g \iff \overline{g}]] \wedge \\
&\qquad [\mathsf{Error} \iff \overline{\mathsf{Error}}]
\end{aligned}
$$

Figure 22: Predicates Describing the Semantics of the Language from Figure

$$
\begin{array}{rcll}
F & - & \text{a role logic formula} & \\
A & - & \text{unary predicate} & \\
f & - & \text{binary predicate} & \\
\mathsf{procedure} & ::= & \mathsf{procName}(\mathsf{unaryList}) = \mathsf{stat} & \\
\mathsf{refinement} & ::= & \mathsf{procName} \Rightarrow \mathsf{procName} & \\
\mathsf{unaryList} & ::= & A \mid \mathsf{unaryList}, A & \\
\mathsf{stat} & ::= & \mathsf{asgnStat} & \text{assignment statement} \\
& \mid & \mathsf{procName}(\mathsf{paramList}) & \text{procedure call} \\
& \mid & \mathsf{assume}\, F & \text{assume statement} \\
& \mid & \mathsf{assert}\, F & \text{assert statement} \\
& \mid & \mathsf{spec}\, F_E & \text{specification} \\
& \mid & \mathsf{stat} \vee \mathsf{stat} & \text{non-deterministic choice} \\
& \mid & \mathsf{stat} \wedge \mathsf{stat} & \text{conjunction} \\
& \mid & \mathsf{stat}; \mathsf{stat} & \text{sequential composition} \\
\mathsf{asgnStat} & ::= & A := F & \text{update of unary predicate} \\
& \mid & F_1.f := F_2 & \text{update of binary predicate} \\
& \mid & F_1.{\sim}f := F_2 & \text{update of inverse of binary predicate} \\
F_E & ::= & A \mid f \mid \mathsf{EQ} \mid F_1 \wedge F_2 \mid \neg F & \\
& \mid & F' \mid {\sim}F \mid \mathsf{card}^{\geq k} F & \\
& \mid & \mathsf{asgnStat} \mid \mathsf{modify\ items} \mid \mathsf{procName}(\mathsf{paramList}) & \\
\mathsf{paramList} & ::= & F \mid \mathsf{paramList}, F & \\
\mathsf{items} & ::= & \mathsf{modItem} \mid \mathsf{items}, \mathsf{modItem} & \\
\mathsf{modItem} & ::= & A :<= F & \text{modification of unary predicate} \\
& \mid & F_1.f :<= F_2 & \text{modification of binary predicate} \\
& \mid & F_1.{\sim}f :<= F_2 & \text{modification of inverse of binary predicate}
\end{array}
$$

Figure 21: Syntax of a Small Imperative Language

```
proc assignClients() =
spec old(GlobalInvariant) =>
 (modify WaitingClients, AssignedClients,
         old(WaitingClients).server :<= Servers,
         Servers.clients :<= old(WaitingClients)) &
 !{WaitingClients} &
 [AssignedClients <=>
           old(AssignedClients | WaitingClients)] &
 GlobalInvariant

proc assignOneClient(cl) =
spec old(GlobalInvariant) &
     [cl => old(WaitingClients)] =>
 (modify WaitingClients, AssignedClients,
         cl.server :<= Servers,
         Servers.clients :<= cl) &
 [WaitingClients | cl <=> old(WaitingClients)] &
 [AssignedClients <=> old(AssignedClients) | cl] &
 GlobalInvariant
```

Figure 24: Specifications for `assignClients` and `assignOneClient` extended with side effect specifications.

occurred (the sequential composition operator ensures that the Error value is propagated).

The statement spec F_E allows describing relations on states directly in terms of an extended RL^2 formula F_E. Formula F_E allows assignment statements and modifies statements in addition to the constructs of RL^2. The relation symbols of RL^2 may refer to relation symbols of the extended language, which allows stating relations between pre and postcondition. We also allow non-recursive procedure calls in the specification when they expand to constructs not containing sequential composition.

modify specifications. The construct

$$\mathsf{modify}\ e_1, \ldots, e_n$$

is useful for specifying frame conditions. Each expression e_i specifies a set of possible modifications. Any finite number of modifications can occur as the result of the action specified by the modify specification.

Example 8 Figure 24 shows the specifications `assignClients` and `assignOneClient` from Figure 3 extended with frame-condition specifications. The frame condition for `assignOneClient` specifies that only the sets WaitingClients and AssignedClients can change, which is useful if the system contains some additional set of objects, such as a set ProcessedClients. Next, the frame-condition specifies that the only binary relations that were modified are server and clients. The modifies expression (Servers.clients :<= cl) indicates that the the only way in which the clients relation is changed is by introducing an edge from a Servers object to the cl object, or by removing an edge from a Servers object. (The removal of the edge does not, in fact, occur in `assignOneClientIMPL` in Figure 3, but the frame condition is a conservative approximation.) The amount of detail in specifications such as modifies clauses depends on how strong property we need to prove. The strength of the property, in turn, depends either on some high-level program correctness requirement, or on the amount of information we need about the procedure to prove the properties of its callers. In Figure 3, we did not use modify specification for `assignOneClient` because we did not need it to prove the conformance of `assignClientsIMPL` with respect to `assignClients`. However, even in Figure 3 we needed to know that, for example, `getServer` preserves the global invariant, which follows from the fact that it does not modify any sets or relations (the conjunction with skip implies that `getServer` is a pure function).

♦

In general, there are three forms of modification expressions. The expression $A :<= F$ specifies modifications that remove an element from the set A or insert into A an element that satisfies F. For example, after executing the statement

$$\mathsf{modify}\ A :<= F$$

the set A may contain any subset of the set of objects given by the expression $\overline{A} \vee F$. The expression $F_1.f :<= F_2$ specifies modifications that 1) remove a tuple $\langle o_1, o_2 \rangle$ from the relation interpreting the predicate f, when o_1 satisfies F_1, or 2) insert a tuple $\langle o_1, o_2 \rangle$ into the relation interpreting f, when o_1 satisfies F_1 and $\langle o_1, o_2 \rangle$ satisfies F_2. Similarly, $F_1.{\sim}f :<= F_2$ allows removing $\langle o_1, o_2 \rangle$ from the interpretation of f when o_1 satisfies F_1, or inserting $\langle o_1, o_2 \rangle$ when o_2 satisfies F_1 and $\langle o_1, o_2 \rangle$ satisfy ${\sim}F_2$.

If r_i is the relation describing a modification given by the expression e_i, then the meaning of modify $e_1, \ldots, e_n$ is given by the relation

$$(r_1 \cup \ldots \cup r_n)^* \tag{4}$$

where r^* denotes the transitive closure of relation r. The simple semantics (4) provides good intuition about the meaning of modify statement and makes it clear that the modify statement is idempotent [44]. Figure 23 presents an alternative semantics, which directly encodes a modify statement as an RL^2 formula. The advantage of the semantics in Figure 23 is that it eliminates the need for transitive closure of the transition relation.

Disjunction and conjunction. The language allows computing disjunction and conjunction on statements. Disjunction $\vee$ has a natural interpretation as a non-deterministic choice of commands. Conjunction $\wedge$ is useful for combining nondeterministic statements. Logical operations on statements translate directly to the corresponding logical operations on RL^2 formulas.

Computing sequential composition. When encoding sequential composition of statements in RL^2, we introduce copies $L_{(i)}$ of predicate names in L for $i \in \{1, 2, \ldots\}$. These copies of predicate names denote the values of predicates at program points between the initial and the final program state. Because the definition of relation composition $r_1 \circ r_2 = \{\langle x, z \rangle \mid \exists y.\ \langle x, y \rangle \in r_1 \wedge \langle y, z \rangle \in r_2\}$ involves existential quantification over y, we treat the newly introduced predicates as being existentially quantified. The technique of introducing new predicate names allows us to precisely compute relation composition even for non-deterministic commands.

Procedure calls. The meaning of a procedure is also a relation on states, where the initial state is extended with one unary predicate symbol for each parameter name. In the simple translation of Figure 22, a procedure call identifies parameters with the sets that describe their values by

$$
\begin{aligned}
&\mathcal{M}[\![\mathsf{modify}\ e_1,\ldots,e_n]\!] = \\
&\ \textbf{let}\ \{e_1,\ldots,e_n\} = \\
&\ \{A_1 :<= F_1,\ldots,A_k :<= F_k, \\
&\ \ F_{k+1}.f_{k+1} :<= G_{k+1},\ldots,F_l.f_l :<= G_l, \\
&\ \ F_{l+1}.{\sim}f_{l+1} :<= G_{l+1},\ldots,F_m.{\sim}f_m :<= G_m\} \\
&\ \textbf{in} \\
&\ \bigwedge_{A\notin\{A_1,\ldots,A_k\}} [A \Leftrightarrow \overline{A}] \land \\
&\ \bigwedge_{A\in\{A_1,\ldots,A_k\}} [(\neg\overline{A} \land \textstyle\bigwedge_{A_i\equiv A} \neg\overline{F_i}) \Rightarrow \neg A] \land \\
&\ \bigwedge_{f\notin\{f_{k+1},\ldots,f_m\}} [[f \Leftrightarrow \overline{f}]] \land \\
&\ \bigwedge_{f\in\{f_{k+1},\ldots,f_m\}} [[(\bigwedge_{\substack{f_i\equiv f\\ i\leq l}} \neg F_i' \ \land \bigwedge_{\substack{f_i\equiv f\\ l<i}} \neg F_i) \ \Rightarrow \ (f \Leftrightarrow \overline{f})]] \\
&\ \bigwedge_{f\in\{f_{k+1},\ldots,f_m\}} [[(\neg\overline{f} \land \bigwedge_{\substack{f_i\equiv f\\ i\leq l}} \neg(F_i' \land G_i) \land \bigwedge_{\substack{f_i\equiv f\\ l<i}} \neg(F_i \land {\sim}G_i)) \ \Rightarrow \ \neg f]]
\end{aligned}
$$

Figure 23: Semantics of modify statement.

performing the substitution. Substitution suffices to give semantics to procedures because we assume that the recursion is split using refinement claims. Loops are represented as recursive procedures, so we effectively require loop invariants.

Refinement claims. If P_1 and P_2 are procedure names, the refinement claim $P_1 \Rightarrow P_2$ is a proof obligation that the relation given by the body of procedure P_1 is contained in the relation given by the body of P_2. The intended use of the refinement claim is the specification procedure summaries, which allows breaking the cycles in the call graphs of mutually recursive procedures. Figure 22 shows how each refinement claim reduces to a test whether an RL^2 formula is satisfiable. When generating the RL^2 formula, we rename the parameters of P_2 replacing them with the corresponding parameters of P_1.

To ensure that the satisfiability test treats newly introduced predicates as existentially quantified, we impose a restriction that the translation $[\![S_2]\!]$ contains no newly introduced predicates from $L_{(i)}$ for $i \in \{1,2,\ldots\}$. We impose this restriction because $[\![S_2]\!]$ appears under negation in the satisfiability test, so newly introduced predicates in $[\![S_2]\!]$ would be universally quantified, thus violating the semantics of sequential composition for non-deterministic statements. The restriction on S_2 is satisfied when S_2 contains no sequential composition, which is typically the case for a large class of procedure summaries.

By providing sufficiently many procedure summaries, the partial correctness of a program is reduced to a finite number of refinement claims. By discharging these claims using a decision procedure for RL^2, we decide the partial correctness of the program.

Fixpoint computation. If some procedure summaries are not supplied by the programmer, they can be inferred using fixpoint computation. An algorithm for fixpoint computation can be derived from the fixpoint semantics of mutually recursive procedures using abstract interpretation [19, 21, 20, 70]. A special case of this approach is to select a finite subset of all RL^2 formulas and define a lattice structure on the set using the entailment of formulas. A simple way to define a finite subset of formulas is to consider only RL^2 formulas with quantifier depth at most k, for some $k \geq 1$. Boolean shape analysis constraints in Section 5.2 have quantifier depth at most two, so they can be used as a basis of fixpoint computation.

$$
\begin{aligned}
F &::= \{C\} \mid \{\{C_1' \land C_2 \land R\}\} \mid F_1 \land F_2 \mid \neg F \\
C &::= A \mid C_1 \land C_2 \mid \neg C \\
R &::= f \mid \neg f \mid R_1 \lor R_2 \\
A &\ - \ \text{atomic unary predicate} \\
f &\ - \ \text{atomic binary predicate}
\end{aligned}
$$

Figure 25: Boolean Shape Analysis Constraints expressed as a sublogic of RL^2

5.2 Describing Boolean Shape Analysis Constraints

Boolean Shape Analysis Constraints [48] are a natural language for describing dataflow facts of shape analyses [65].

Figure 25 presents the syntax of Boolean Shape Analysis Constraints as a subset of role logic. This presentation of Boolean Shape Analysis Constraints shows that they are a subset of the decidable fragment RL^2 of role logic. In fact, Boolean Shape Analysis Constraints do not use counting quantifiers, so they are already expressible in the two-variable predicate logic L^2 (without counting).

A note on usability of role logic. An anecdotal evidence of the usability of role logic is the fact that all results

$$
\begin{array}{lcl}
C & ::= & A \mid C \sqcap C \mid \neg C \mid \geq nR.C \\
R & ::= & f \mid R \sqcap R \mid \neg R \mid U \mid R^{-1} \mid R_{|C} \mid \mathsf{id}(C) \\
A & - & \text{atomic unary predicate} \\
f & - & \text{atomic binary predicate}
\end{array}
$$

Figure 26: An Expressive Description Logic

$$
\begin{array}{rcl}
[\![A]\!] & = & A \\
[\![C_1 \sqcap C_2]\!] & = & [\![C_1]\!] \wedge [\![C_2]\!] \\
[\![\neg C]\!] & = & \neg [\![C]\!] \\
[\![\geq nR.C]\!] & = & \mathsf{card}^{\geq n}([\![R]\!] \wedge [\![C]\!]) \\
[\![f]\!] & = & f \\
[\![R_1 \sqcap R_2]\!] & = & [\![R_1]\!] \wedge [\![R_2]\!] \\
[\![\neg R]\!] & = & \neg [\![R]\!] \\
[\![U]\!] & = & \mathsf{true} \\
[\![R^{-1}]\!] & = & \sim [\![R]\!] \\
[\![R_{|C}]\!] & = & [\![R]\!] \wedge [\![C]\!] \\
[\![\mathsf{id}(C)]\!] & = & \mathsf{EQ} \wedge [\![C]\!]
\end{array}
$$

Figure 27: Translation of an Expressive Description Logic to Role Logic with Two Variables

of [48] were initially shown using role logic notation and then translated into the standard first-order logic notation. We have found the variable-free aspect of role logic convenient when showing the results of [48]. We have subsequently discovered the connection of role logic with C^2 [30], presented in Section 4, and the connection with description logics [1], presented in Section 5.3.

5.3 Encoding an Expressive Descriptive Logic

Figure 26 presents an Expressive Description Logic fragment where roles have no transitive operators [1, Chapter 5]. Figure 27 presents the translation of the Expressive Description Logic into RL^2. The translation maps the concepts C and roles R of description logic into unary and binary predicates of role logic. The translation to RL^2 in Figure 27 implies that the description logic in Figure 26 is decidable. The fact that interesting description logics can be translated to RL^2 is not surprising once we have established that RL^2 and C^2 have equal expressive power. Nevertheless, it is interesting to observe the simplicity of the translation from the description logic to RL^2, which is partly because both description logic and role logic avoid explicit occurrences of variables.

Using rules

$$
\begin{array}{rcl}
[\![R_1 \circ R_2]\!] & = & [\![R_1]\!] \circ [\![R_2]\!] \\
[\![R^*]\!] & = & [\![R]\!]^*
\end{array}
$$

we can translate operations on binary relations into the full role logic, but not into the decidable fragment RL^2. Decidability of interesting description logics that contain transitive closure but do not have tree model property is an open problem [1, Page 214].

A note on terminology. The term "role" has different meanings in different formalisms for describing structures. In [43], a role corresponds to a unary predicate (set), in description logics [1], a role corresponds to a binary predicate (relation), and in entity-relationship diagrams in databases [16], a role corresponds to a position i ($1 \leq i \leq n$) in a n-tuples of an n-ary relation. To avoid the confusion, we use the well-established terms of n-ary "predicate" (or "relation"), keep the name "role logic" for the logic described in Figure 9, because the term "role logic" appears appropriate regardless of the particular interpretation of the word "role".

Description Logics Corresponding to C^2. [1] The result [10, Theorem 4] reports that the description logic without transitive closure and relation composition (denoted $\mathcal{DL}-\{\mathbf{trans}, \mathbf{compose}\}$) corresponds precisely to C^2. The results of Section 4 and [10] imply that our logic RL^2 has the same expressive power as $\mathcal{DL}-\{\mathbf{trans}, \mathbf{compose}\}$. One of the differences between RL^2 and $\mathcal{DL}-\{\mathbf{trans}, \mathbf{compose}\}$ is that RL^2 contains the prime operator F' and does not contain the **product** operation of $\mathcal{DL}-\{\mathbf{trans}, \mathbf{compose}\}$. Another difference is the foundation of role logic on de Bruijn lambda calculus notation, as described in Section 3.

6 Related Work

We have initially developed role logic to provide a foundation for role analysis [43, 42]. We have subsequently studied a simplification of role analysis constraints and showed a characterization of such constraints using formulas [46]. Parametric analysis based on three-valued logic was introduced in [64, 65] with interprocedural analysis in [61] and application to abstract data type verification in [52]. A characterization of dataflow facts used for shape analysis was presented in [71, 48]. A decidable logic for expressing connectivity properties of the heap was presented in [7].

Specifying the semantics of programs using predicates dates back to axiomatic program semantics [32, 24]. An approach that uses a first-order logic theorem prover tailed for program verification is [23].

Like [40, 39, 37, 55], in Section 5.1 we use an expressive yet decidable logic to encode fragments of straight-line code. Our approach differs primarily in using logic RL^2 over general graphs whose decidability follows from the decidability of C^2, where [40, 39, 37, 55] uses graph types whose decidability follows from the decidability of monadic second-order logic over trees. We expect that these two logics can be combined in a fruitful way.

We have extended our language with constructs that make it possible to directly express higher-level state transformations, which is the idea related to the chemical reaction model of [26, 27], the verification of database transactions [6], the simultaneous assignments of [55], and in wide-spectrum languages [56, 3]. Verification of a form of modifies clauses using a theorem prover was presented [50, 44]. Further approaches to pointer and shape analysis include [17, 68, 15, 29, 25, 28, 69].

[1] Note added on 31 October 2003, after becoming aware of [10].

Description logics [1, 9] share many of the properties of role logic and have been traditionally applied to knowledge bases. It is likely that description logics can be used for shape analysis as well. It would be particularly interesting to consider description logics with transitive operators, whose decidability is related to the decidability of dynamic logic [31]. Reasoning about the satisfiability of expressive description logics over all structures and over finite structures is presented in [13, 14]. Reasoning about entity-relationship diagrams [16] is presented in [51]. Some connections between object models and heap invariants are presented in [45, 35].

Like the Alloy modelling language [36], role logic combines the notation of predicate calculus with the notation of relational algebras. It may be possible to combine the notation of Alloy with the notation of role logic, and to combine the benefits of bounded model checking used in Alloy Analyzer with the benefits of a decision procedure for RL^2.

A recent approach to reasoning about mutable imperative data structure is separation logic [34, 59, 60, 12, 11]. We are currently working on integrating some aspects of spatial logic to support more flexible notation for records in role logic.

Interactive theorem provers have also been used for reasoning about dynamically allocated data structures [54, 2]; it may be interesting to incorporate a decision procedure for RL^2 into these general tools.

7 Conclusions

We believe that role logic notation is a convenient way of expressing properties of first-order structures. First-order structures are a natural way to model the state in object-oriented programs, or a the state of a knowledge base or a database. Role logic can be combined with traditional variable-based notation in a natural way. Furthermore, interesting subsets of role logic are decidable. Decision procedures for role logic can therefore enable shape analysis of programs and have similar benefits as description logics in knowledge bases.

Acknowledgements We thank Patrick Lam for useful discussions, comments on the paper, and an implementation of an early version of role logic normalization algorithm in Fall 2001, we thank Andreas Podelski for discussion of using formulas to perform shape analysis, we thank Thomas Reps for discussions on summarizing procedures using two-vocabulary structures, we thank C. Scott Ananian for discussion of a draft of this paper in Spring 2003, we thank Derek Rayside, Mooly Sagiv, and Greta Yorsh for useful discussions, and Darko Marinov for comments on the paper.

References

[1] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[2] Ralph-Johan Back, Xiaocong Fan, and Viorel Preoteasa. Reasoning about pointers in refinement calculus. In *10th Asia-Pacific Software Engineering Conference (APSEC'03)*, 2003.

[3] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus*. Springer-Verlag, 1998.

[4] Henk P. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics*. North-Holland, 2nd edition, 1984.

[5] Henk P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Vol. II.* Oxford University Press, 2001.

[6] Michael Benedikt, Timothy Griffin, and Leonid Libkin. Verifiable properties of database transactions. *Information and Computation*, 147:57–88, 1998.

[7] Michael Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for linked data structures. In *Proc. 8th ESOP*, 1999.

[8] Egon Börger and Robert Stärk. *Abstract State Machines*. Springer-Verlag, 2003.

[9] Alexander Borgida. Description logics in data management. *IEEE Trans. on Knowledge and Data Engineering*, 7(5):671–682, 1995.

[10] Alexander Borgida. Description logics in data management. *Artificial Intelligence*, 82(1-2):353–367, 1996.

[11] Cristiano Calcagno, Luca Cardelli, and Andrew D. Gordon. Deciding validity in a spatial logic for trees. In *ACM TLDI'02*, 2002.

[12] Cristiano Calcagno, Samin Ishtiaq, and Peter W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in hoare logic. In *Proc. 2nd International Conference on Principles and Practice of Declarative Programming*, 2000.

[13] Diego Calvanese. Finite model reasoning in description logics. In *Proc. of the 5th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'96)*, pages 292–303. Morgan Kaufmann, 1996.

[14] Diego Calvanese. *Unrestricted and Finite Model Reasoning in Class-Based Representation Formalisms*. PhD thesis, Dipartimento di Informatica e Sistemistica, Universita di Roma "La Sapienza", 1996.

[15] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. ACM PLDI*, 1990.

[16] Peter Pin-Shan Chen. The entity-relationship model–toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[17] Stephen Chong and Radu Rugina. Static analysis of accessed regions in recursive data structures. In *Proc. 10th SAS*, volume 2694 of *LNCS*. Springer, 2003.

[18] Edgar F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[19] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, 1977.

[20] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, pages 237–277. North-Holland, 1977.

[21] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proc. 6th POPL*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[22] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag. Math.*, 34:381–392, 1972.

[23] Cormac Flanagan, K. Rustan M. Leino, Mark Lilibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proc. ACM PLDI*, 2002.

[24] Robert W. Floyd. Assigning meanings to programs. In *Proc. Amer. Math. Soc. Symposia in Applied Mathematics*, volume 19, pages 19–31, 1967.

[25] Pascal Fradet, Ronan Gaugne, and Daniel Le Metayer. An inference algorithm for the static verification of pointer manipulation. Technical Report 980, IRISA, 1996.

[26] Pascal Fradet and Daniel Le Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997.

[27] Pascal Fradet and Daniel Le Métayer. Structured gamma. *Science of Computer Programming, SCP, 31(2-3), pp. 263-289*, 1998.

[28] R. Gaugne, P. Fradet, and D. Le Métayer. Static detection of pointer errors: an axiomatisation and a checking algorithm. In *Proc. European Symposium on Programming, ESOP'96*, LNCS, 1996.

[29] Rakesh Ghiya and Laurie Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996.

[30] Erich Grädel, Martin Otto, and Eric Rosen. Two-variable logic with counting is decidable. In *Proceedings of 12th IEEE Symposium on Logic in Computer Science LICS '97, Warschau*, 1997.

[31] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, Cambridge, Mass., 2000.

[32] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[33] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proc. ACM PLDI*, 1994.

[34] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM POPL*, 2001.

[35] Daniel Jackson. Object models as heap invariants. In Annabelle McIver and Carroll Morgan, editors, *Collected Papers of IFIP Working Group 2.3 on Programming Methodology*. Springer-Verlag, 2001.

[36] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM TOSEM*, 11(2):256–290, 2002.

[37] Jacob L. Jensen, Michael E. Jørgensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second order logic. In *Proc. ACM PLDI*, Las Vegas, NV, 1997.

[38] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[39] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proc. 20th ACM POPL*, Charleston, SC, 1993.

[40] Nils Klarlund and Michael I. Schwartzbach. Graphs and decidable transductions based on edge constraints. In *Proc. 19th Colloquium on Trees and Algebra in Programming*, number 787 in LNCS, 1994.

[41] Robert Kowalski. Algorithm = logic + control. *Communications of the ACM*, 1979.

[42] Viktor Kuncak. Designing an algorithm for role analysis. Master's thesis, MIT Laboratory for Computer Science, 2001.

[43] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proc. 29th POPL*, 2002.

[44] Viktor Kuncak and K. Rustan M. Leino. In-place refinement for effect checking. In *Second International Workshop on Automated Verification of Infinite-State Systems (AVIS'03), Warsaw, Poland*, April 2003.

[45] Viktor Kuncak and Martin Rinard. Object models, heaps, and interpretations. Technical Report 816, MIT Laboratory for Computer Science, January 2001.

[46] Viktor Kuncak and Martin Rinard. Typestate checking and regular graph constraints. Technical Report 863, MIT Laboratory for Computer Science, 2002.

[47] Viktor Kuncak and Martin Rinard. Existential heap abstraction entailment is undecidable. In *10th Annual International Static Analysis Symposium (SAS 2003)*, San Diego, California, June 11-13 2003.

[48] Viktor Kuncak and Martin Rinard. On the boolean algebra of shape analysis constraints. Technical report, MIT CSAIL, August 2003.

[49] Viktor Kuncak and Martin Rinard. On the theory of structural subtyping. Technical Report 879, Laboratory for Computer Science, Massachusetts Institute of Technology, 2003.

[50] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proc. ACM PLDI*, 2002.

[51] Maurizio Lenzerini and Paolo Nobili. On the satisfiability of dependency constraints in entity-relationship schemata. In *Proc. 13th VLDB*, pages 147–154, 1987.

[52] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis*, 2000.

[53] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[54] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, LNCS. Springer-Verlag, 2003.

[55] Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. ACM PLDI*, 2001.

[56] Carroll Morgan. *Programming from Specifications (2nd ed.)*. Prentice-Hall, Inc., 1994.

[57] Leszek Pacholski, Wieslaw Szwast, and Lidia Tendera. Complexity results for first-order two-variable logic with counting. *SIAM J. on Computing*, 29(4):1083–1117, 2000.

[58] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer-Verlag, 1994.

[59] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Proceedings of the Symposium in Celebration of the Work of C.A.R. Hoare*, 2000.

[60] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17th LICS*, pages 55–74, 2002.

[61] Noam Rinetzky and Mooly Sagiv. Interprocedual shape analysis for recursive programs. In *Proc. 10th International Conference on Compiler Construction*, 2001.

[62] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations Vol.1*. World Scientific, 1997.

[63] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, Reading, Mass., 1999.

[64] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. 26th ACM POPL*, 1999.

[65] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[66] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in Setl programs. *Transactions on Programming Languages and Systems*, 3(2):126–143, 1991.

[67] Thoralf Skolem. Untersuchungen über die Axiome des Klassenkalküls and über "Produktations- und Summationsprobleme", welche gewisse Klassen von Aussagen betreffen. Skrifter utgit av Vidnskapsselskapet i Kristiania, I. klasse, no. 3, Oslo, 1919.

[68] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Denver, November 1999.

[69] R. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. ACM PLDI*, June 1995.

[70] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *20th ACM POPL*, 1993.

[71] Greta Yorsh. Logical characterizations of heap abstractions. Master's thesis, Tel-Aviv University, March 2003.