A Decidable Logic for Describing Linked Data Structures

Michael Benedikt[1], Thomas Reps[2], and Mooly Sagiv[3]

[1] Bell Laboratories, Lucent Technologies, benedikt@research.bell-labs.com
[2] University of Wisconsin, reps@cs.wisc.edu
[3] Tel-Aviv University, sagiv@math.tau.ac.il.

Abstract. This paper aims to provide a better formalism for describing properties of linked data structures (e.g., lists, trees, graphs), as well as the intermediate states that arise when such structures are destructively updated. The paper defines a new logic that is suitable for these purposes (called L_r, for "logic of reachability expressions"). We show that L_r is decidable, and explain how L_r relates to two previously defined structure-description formalisms ("path matrices" and "static shape graphs") by showing how an arbitrary shape descriptor from each of these formalisms can be translated into an L_r formula.

1 Introduction

This paper aims to provide a better formalism for describing properties of linked data structures (e.g., lists, trees, graphs). In past work with the same motivation, a variety of different formalisms have been developed — including "static shape graphs" [14, 15, 17, 12, 3, 23, 1, 19, 27, 21, 20, 22], "path matrices" [9, 11], "graph types" [16], and the ADDS annotation formalism [10] — and several previously known formalisms have been exploited — including graph grammars [6] and monadic second-order logic [13]. For lack of a better term, we will use the phrase *structure-description formalisms* to refer to such formalisms in a generic sense.

In this paper, we define a new logic (called L_r, for "logic of reachability expressions"), and show that L_r is suitable for describing properties of linked data structures. We show that L_r is decidable. We also show in detail how L_r relates to two of the previously defined structure-description formalisms: In Section 3, we show how a generalization of Hendren's path-matrix descriptors [9, 11] can be represented by L_r formulae; in Section 4, we show how the variant of static shape graphs defined in [21] can be represented by L_r formulae. In this way, L_r provides insight into the expressive power of path matrices and static shape graphs.

The benefits of our work include the following:

– The logic L_r can be used as an annotation language to express loop invariants and pre- and post-conditions of statements and procedures. Annotations are important not only as a means of documenting programs, but also as the basis for analyzing and reasoning about programs in a modular fashion. Our work has two advantages:

- The logic L_r is quite expressive (e.g., strictly more expressive than the formalism used by Hendren et al. [10]). The added expressibility is important for describing the intermediate states that arise when linked data structures are destructively updated.
- The logic L_r is *decidable*, which means that there is an algorithm that determines, for every formula in the logic, if the formula is satisfiable. In other words, it is possible to determine if there is any store at all that satisfies a given formula. In principle, this ability can be used to provide some sanity checks on the formulae that a user employs — e.g., a warning can be issued if the user employs a formula that is unsatisfiable.

– Our work makes contributions on the question of *extracting information from the results of program analysis*. Although the subject of the paper is not primarily *algorithms for analyzing* programs that manipulate linked data structures, the decidability of L_r — together with the constructions given in Sections 3 and 4 for encoding other structure-description formalisms in L_r — has interesting consequences for extracting information from the results of program analyses: L_r provides a way to *amplify* the results obtained from known pointer-analysis, alias-analysis, and shape-analysis algorithms in the following ways:
  - For a structure-description formalism in which each structure descriptor corresponds to an L_r formula, as is the case for path matrices (Section 3) and static shape graphs (Section 4), it is possible to determine if there is any store at all that corresponds to a given structure descriptor. This lets us determine whether a given structure descriptor contains any useful information.
  - Pointer-analysis, alias-analysis, and shape-analysis algorithms necessarily compute structure descriptors that over-approximate the pointer/alias/shape relationships that actually arise. This kind of loss of precision is intrinsic to static-analysis; however, many of the techniques that have been proposed in the literature have the feature that *additional imprecision* crops up when information is extracted from the structure descriptor for a particular program point. For instance, with the three-valued logic used for shape analysis in [20, 22], a formula that queries for a specific piece of information sometimes evaluates to "unknown", even when, in all of the stores that the static shape graph represents, the formula evaluates to a definite true or false value.

    For a structure-description formalism in which each structure descriptor corresponds to an L_r formula, decidability gives us a mechanism for *reading out information obtained by existing algorithms, without any additional loss of precision*: If φ is the formula that represents the shape descriptor and ψ is the formula that represents the query, we are interested in whether $\varphi \implies \psi$ always holds (or, equivalently, whether $\neg(\varphi \implies \psi)$ is unsatisfiable). Thus, in principle, the machinery developed in this paper allows us to take the structure descriptors computed by existing techniques, and extract information from them that is more precise than that envisioned by the inventors of these formalisms.

- For many of the structure-description formalisms used in the literature, very little is known about basic decision problems associated with them. Mapping a structure-description formalism F into L_r can provide a way to analyze many basic decision problems of F.

  For instance, a decision problem of special interest for structure-description formalisms that are used in abstract interpretation is the inclusion problem (i.e., whether the set of stores that structure descriptor D_1 represents is a subset of the set of stores that D_2 represents). When the inclusion problem is decidable, it is possible to check (i) whether one structure descriptor subsumes another (and hence the second need not be retained), and (ii) whether a simpler structure descriptor is a conservative replacement of a larger one, which is useful in widening. Thus, the inclusion problem is important for reducing both the time and space used during abstract interpretation.

  For a structure-description formalism in which each structure descriptor corresponds to an L_r formula, the inclusion of structure descriptor D_1 (represented by formula φ_1) in D_2 (represented by φ_2) is a matter of testing whether $\varphi_1 \implies \varphi_2$ always holds (or, equivalently, whether $\neg(\varphi_1 \implies \varphi_2)$ is unsatisfiable).

To date, our concern has been with developing the tools for describing properties of linked data structures and obtaining a logic that is decidable. We have developed a decision procedure for L_r, although this procedure does not yield a practical algorithm. We have not yet investigated the complexity of the decision problem for L_r, nor looked for heuristic methods with acceptable performance in practice, but we plan to do so in future work.

Two programs that will be used to illustrate our work are shown in Figure 1. The remainder of the paper is organized into six sections: Section 2 presents the logic we use for describing properties of linked data structures. Section 3 shows how a generalization of Hendren's path-matrix descriptors [9, 11] can be represented by L_r formulae. Section 4 shows how a variant of static shape graphs can be represented by L_r formulae. Section 5 discusses the issue of using L_r formulae to extract information from the results of program analyses. Section 6 gives a sketch of the proof that L_r is decidable. Section 7 discusses related work.

2 A Language for Stating Properties of Linked Data Structures

Definition 21 *Let $PVar$ be the (finite) set of pointer variables in a given program. Let Sel be the set of* **pointer selectors** *(i.e., pointer-valued fields of structures) used in the program. We define the* **alphabet** *Σ to be the following finite set of symbols: $\Sigma = Sel \cup \{\texttt{pvar}? \mid \texttt{pvar} \in PVar\} \cup \{\neg\texttt{pvar}? \mid \texttt{pvar} \in PVar\}$, with the intended meaning that* `pvar`? *denotes the cell pointed to by the pointer variable* `pvar`*, and* $\neg$`pvar`? *denote cells not pointed to by* `pvar`. *A* **formula** *in*

```
typedef struct elem_list_struct {
        int val;
        struct elem_list_struct *cdr; } Elements;
```
(a)

```
Elements * elem_reverse(Elements *x
/* acyclic_list(x) */)
{ Elements *z, *y;
  y = NULL ;
  while (x != NULL) {
        /* acyclic_list(x) */
        /* acyclic_list(y) */
        /* disjoint_lists(x,y) */
        z = y;
        y = x ;
        x = x →cdr;
        y →cdr = z; }
  /* acyclic_list(y) */
  return y ;
}
```
(b)

```
bool elem_delete(int delval, Elements *c)
{ /* acyclic_list(c) */
  Elements *elem,*prev;
  for (prev = NULL, elem = c;
       elem != NULL;
       prev=elem, elem = elem →cdr) {
     if (elem →val == delval) {
       if (prev == NULL)
          c = elem →cdr;
       else
            prev →cdr = elem →cdr;
       free(elem);
       return TRUE; } }
  /* acyclic_list(c) */
  return FALSE;
}
```
(c)

Fig. 1. (a) A C declaration of a linked-list type. (b)A program that uses destructive-updating operations to reverse a list. (c)A program that searches the list pointed to by variable `c` (using a "trailing pointer" `prev`) and deletes the first element whose `val` field equals `delval`.

the logic L_r is defined as follows:

$$
\begin{array}{rlr@{\qquad}rlr}
\Phi ::= & \mathtt{pe}_1 = \mathtt{pe}_2 & \textit{equality of pointer exps.} & R ::= & \epsilon & \textit{empty path} \\
\mid & \mathtt{pe}_1\langle R\rangle\mathtt{pe}_2 & \textit{reachability constraint} & \mid & \emptyset & \textit{empty lang.} \\
\mid & hs(\mathtt{pe}\langle R\rangle) & \textit{heap-sharing constraint} & \mid & \sigma & \sigma \in \Sigma \\
\mid & al(\mathtt{pe}\langle R\rangle) & \textit{allocation constraint} & \mid & R_1.R_2 & \textit{concat.} \\
\mid & \neg\Phi & \textit{negation} & \mid & R_1|R_2 & \textit{union} \\
\mid & \Phi_1 \wedge \Phi_2 & \textit{conjunction} & \mid & R^* & \textit{Kleene star} \\
\mid & \Phi_1 \vee \Phi_2 & \textit{disjunction} & \mathtt{pe} ::= & \mathtt{pvar} & \textit{pointer var.} \\
\mid & \Phi_1 \implies \Phi_2 & \textit{implication} & \mid & \mathtt{pe.sel} & \mathtt{sel} \in \textit{Sel}
\end{array}
$$

We call R terms **routing expressions***, and refer to occurrences of* `pvar`*? and* ¬`pvar`*? in routing expressions as* **pointer-variable interrogations***.*

We also use several shorthand notations: $hs(\mathtt{p})$ and $al(\mathtt{p})$ are shorthands for $hs(\mathtt{p}\langle\epsilon\rangle)$ and $al(\mathtt{p}\langle\epsilon\rangle)$, respectively. Similarly, $hs(\mathtt{p.sel})$ and $al(\mathtt{p.sel})$ are shorthands for $hs(\mathtt{p}\langle\mathtt{sel}\rangle)$ and $al(\mathtt{p}\langle\mathtt{sel}\rangle)$, respectively. $\mathtt{pe}_1 \neq \mathtt{pe}_2$ is a shorthand for $\neg(\mathtt{pe}_1 = \mathtt{pe}_2)$. $\Phi_1 \Leftrightarrow \Phi_2$ is a shorthand for $(\Phi_1 \implies \Phi_2) \wedge (\Phi_2 \implies \Phi_1)$.

Example 22 For a pointer variable `x`, the formula

$$\textit{acyclic\_list}(\mathtt{x}) \stackrel{\text{def}}{=} \neg\mathtt{x}\langle\mathtt{cdr}^+\rangle\mathtt{x} \wedge \neg hs(\mathtt{x}\langle\mathtt{cdr}^*\rangle)$$

states that x points to an unshared acyclic list. The term $\neg\mathtt{x}\langle\mathtt{cdr}^+\rangle\mathtt{x}$ signifies that a traversal that starts with the cell pointed to by x and follows one or more cdr fields never returns to the cell pointed to by x. The term $\neg hs(\mathtt{x}\langle\mathtt{cdr}^*\rangle)$ signifies that a traversal that starts with the cell pointed to by x and follows zero or more cdr fields never leads to a cell that is "heap shared". (A cell c is "heap shared" if two or more cells have fields that point to c, or if there is a cell c' such that c'.car and c'.cdr both point to c [15, 3, 21, 20].)

Thus, a loop invariant for program elem_reverse can be written as follows:

$$\begin{array}{l} \quad((al(\mathtt{y.cdr}) \vee al(\mathtt{z})) \implies \mathtt{y.cdr} = \mathtt{z}) \\ \wedge\ acyclic\_list(\mathtt{x}) \wedge acyclic\_list(\mathtt{y}) \\ \wedge\ \neg\mathtt{x}\langle\mathtt{cdr}^*\rangle\mathtt{y} \wedge \neg\mathtt{x}\langle\mathtt{cdr}^*\rangle\mathtt{z} \wedge \neg\mathtt{y}\langle\mathtt{cdr}^*\rangle\mathtt{x} \wedge \neg\mathtt{z}\langle\mathtt{cdr}^*\rangle\mathtt{x} \end{array} \tag{1}$$

The first line of (1) states that y.cdr and z refer to the same list element when either one is allocated. The subformulae on the last line of (1) states that the x-list is disjoint from both the y-list and the z-list.

Example 23 A loop invariant for program elem_delete can be written as follows:

$$\begin{array}{l} \quad acyclic\_list(\mathtt{c}) \wedge \mathtt{c}\langle\mathtt{cdr}^*\rangle\mathtt{elem} \\ \wedge\ al(\mathtt{prev}) \implies (\mathtt{c}\langle\mathtt{cdr}^*\rangle\mathtt{prev} \wedge \mathtt{prev.cdr} = \mathtt{elem}) \\ \wedge\ \neg al(\mathtt{prev}) \iff \mathtt{elem} = \mathtt{c} \end{array} \tag{2}$$

The subformula $\mathtt{c}\langle\mathtt{cdr}^*\rangle\mathtt{elem}$ states that elem points somewhere in the list pointed to by c. The subformula on the last line of (2) states that prev is allocated (i.e., not NULL) if and only if elem and c point to different locations. From this, we can conclude that the location released by the statement free(elem) cannot be pointed to by c.

The use of pointer-variable interrogations in routing expressions will be illustrated in Examples 33 and 36.

We now define the semantics of L_r formulae:

Definition 24 *A store S can be represented by a tuple $\langle Loc^S, env^S, \iota^S\rangle$, where Loc^S is a set of locations, and env^S and ι^S are functions*

$$\begin{array}{rcl} env^S & : & PVar \to (Loc^S \cup \{\bot\}) \\ \iota^S & : & Sel \to (Loc^S \cup \{\bot\}) \to (Loc^S \cup \{\bot\}), \end{array}$$

where ι^S is strict in $\bot$.

The meaning of a pointer expression pe *in a given store S, denoted by $[\![\mathtt{pe}]\!]^S$ (where $[\![\mathtt{pe}]\!]^S \in (Loc^S \cup \{\bot\})$), is defined inductively, as follows:*

$$\begin{array}{rcl} [\![\mathtt{pvar}]\!]^S & = & env^S(\mathtt{pvar}) \\ [\![\mathtt{pe.sel}]\!]^S & = & \iota^S(\mathtt{sel})([\![\mathtt{pe}]\!]^S) \end{array}$$

The language $L(R)$ of a routing expression R is defined as is usual for regular expressions. However, because a word in $L(R)$ can contain occurrences of pointer-variable interrogations of the form pvar? *and* $\neg$pvar?, *the meaning of a word is* *slightly nonstandard: The meaning of a word in a given store S, denoted by $[\![w]\!]^S$ (where $[\![w]\!]^S : (Loc^S \cup \{\bot\}) \rightarrow (Loc^S \cup \{\bot\})$), is defined inductively, as follows:*

$$
\begin{aligned}
[\![\epsilon]\!]^S(l) &= l \\
[\![w.\texttt{sel}]\!]^S(l) &= \iota^S(\texttt{sel})([\![w]\!]^S(l)) \\
[\![w.\texttt{pvar}?]\!]^S(l) &= \begin{cases} [\![w]\!]^S(l) & \text{if } env(\texttt{pvar}) = [\![w]\!]^S(l) \\ \bot & \text{otherwise} \end{cases} \\
[\![w.\neg\texttt{pvar}?]\!]^S(l) &= \begin{cases} [\![w]\!]^S(l) & \text{if } env(\texttt{pvar}) \neq [\![w]\!]^S(l) \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

The meaning of formula in a given store S is defined inductively, as follows:

$$
\begin{aligned}
[\![\texttt{pe}_1 = \texttt{pe}_2]\!]^S &= ([\![\texttt{pe}_1]\!]^S = [\![\texttt{pe}_2]\!]^S) \\
[\![\texttt{pe}_1\langle R\rangle\texttt{pe}_2]\!]^S &= \text{there exists } w \in L(R) \text{ s.t. } [\![w]\!]^S([\![\texttt{pe}_1]\!]^S) = [\![\texttt{pe}_2]\!]^S \text{ and } [\![\texttt{pe}_2]\!]^S \in Loc \\
[\![hs(\texttt{pe}\langle R\rangle)]\!]^S &= \text{there exists } w \in L(R) \text{ s.t. } [\![w]\!]^S([\![\texttt{pe}]\!]^S) = l \text{ and } l \in Loc \text{ and} \\
&\quad\ \text{there exist } l_1, l_2 \in Loc, \texttt{sel}_1, \texttt{sel}_2 \in Sel \text{ s.t. } \iota^S(\texttt{sel}_1)(l_1) = l \text{ and} \\
&\quad\ \iota^S(\texttt{sel}_2)(l_2) = l \text{ and either (i) } l_1 \neq l_2 \text{ or (ii) } \texttt{sel}_1 \neq \texttt{sel}_2 \\
[\![al(\texttt{pe}\langle R\rangle)]\!]^S &= \text{there exists } w \in L(R) \text{ such that } [\![w]\!]^S([\![\texttt{pe}]\!]^S) \in Loc \\
[\![\neg\Phi]\!]^S &= [\![\Phi]\!]^S \text{ is false} \\
[\![\Phi_1 \wedge \Phi_2]\!]^S &= [\![\Phi_1]\!]^S \text{ is true and } [\![\Phi_2]\!]^S \text{ is true} \\
[\![\Phi_1 \vee \Phi_2]\!]^S &= [\![\Phi_1]\!]^S \text{ is true or } [\![\Phi_2]\!]^S \text{ is true} \\
[\![\Phi_1 \implies \Phi_2]\!]^S &= [\![\neg\Phi_1]\!]^S \text{ is true or } [\![\Phi_2]\!]^S \text{ is true}
\end{aligned}
$$

3 Representing Path Matrices via Formulae

In this section, we study the relationship between the logic L_r and a variant of the *path-matrix* structure-description formalism [9, 11]. A path matrix records information about the (possibly empty) set of paths that exist between pairs of pointer variables in a program. The version of path matrices described below is a generalization of the original version described in [9, 11]. We show that every path matrix (of the extended version of the formalism) can be represented by a formula in logic L_r.

Definition 31 *A path matrix pm contains an entry $pm[\mathtt{x}, \mathtt{y}]$ for every pair of pointer-valued program variables,* x *and* y. *An entry $pm[\mathtt{x}, \mathtt{y}]$ describes the set of paths from the cell pointed to by* x *to the cell pointed to by* y. *An entry $pm[\mathtt{x}, \mathtt{y}]$ has a value of the form $\langle R, Q\rangle$, where R is a regular expression over Σ, and Q is either "P" (standing for "possible path") or "D" (standing for "definite" path).*

The notions of "possible paths" and "definite paths" are somewhat subtle (and the names "possible paths" and "definite paths", which we have adopted from [9, 11], are somewhat misleading). In the discussion below, let S be a store that path matrix pm represents, and let $Paths_S(\mathtt{x}, \mathtt{y})$ denote the set of paths from the cell pointed to by program variable `x` to the cell pointed to by `y`.

– An entry $pm[\mathtt{x}, \mathtt{y}]$ that has the value $\langle R_D, D\rangle$ means that there is a path p in S from the cell pointed to by program variable `x` to the cell pointed to by `y`, such that $p \in L(R_D)$. In other words,

$$Paths_S(\mathtt{x}, \mathtt{y}) \cap L(R_D) \neq \emptyset. \tag{3}$$

Note that only *one* of the paths in $L(R_D)$ need be a path in $Paths_S(\mathtt{x}, \mathtt{y})$ for $pm[\mathtt{x}, \mathtt{y}] = \langle R_D, D\rangle$ to be satisfied.

– An entry $pm[\mathtt{x}, \mathtt{y}]$ that has the value $\langle R_P, P\rangle$ means that $L(R_P)$ is an over-approximation to the set of paths from the cell pointed to by `x` to the cell pointed to by `y`. In other words,

$$Paths_S(\mathtt{x}, \mathtt{y}) \subseteq L(R_P). \tag{4}$$

An alternative way to think about this is as follows: What we really mean by "R_P represents possible paths in store S" is that $\overline{L(R_P)} = \Sigma^* - L(R_P)$ is a set of *impossible* paths of S: That is, an entry $pm[\mathtt{x}, \mathtt{y}]$ that has the value $\langle R_P, P\rangle$ means that none of the paths from the cell pointed to by `x` to the cell pointed to by `y` are in $\overline{L(R_P)}$. Thus, we have

$$Paths_S(\mathtt{x}, \mathtt{y}) \cap \overline{L(R_P)} = \emptyset. \tag{5}$$

These two ways of looking at things are equivalent, as shown by the following derivation: $Paths_S(\mathtt{x}, \mathtt{y}) \cap \overline{L(R_P)} = \emptyset \implies Paths_S(\mathtt{x}, \mathtt{y}) - L(R_P) = \emptyset \implies Paths_S(\mathtt{x}, \mathtt{y}) \subseteq L(R_P)$.

It is instructive to consider some simple examples of possible path-matrix entries:

– An entry $pm[\mathtt{x}, \mathtt{y}]$ that has the value $\langle \emptyset, P\rangle$ represents the fact that there is no path in S from the cell pointed to by program variable `x` to the cell pointed to by `y`.
– An entry $pm[\mathtt{x}, \mathtt{y}]$ that has the value $\langle \epsilon, D\rangle$ represents the fact that `x` and `y` are must-aliases, i.e., `x` and `y` must point to the same cell in all of the stores that the path matrix represents.
– In contrast, an entry $pm[\mathtt{x}, \mathtt{y}]$ with the value $\langle \epsilon, P\rangle$ represents the fact that `x` and `y` are may-aliases, i.e., `x` and `y` might point to the same cell in some of the stores that the path matrix represents, but it is also possible that in other stores that the path matrix represents, there is no path at all from the cell pointed to by `x` to the cell pointed to by `y`.
– More generally, a value $\langle R, P\rangle$ for entry $pm[\mathtt{x}, \mathtt{y}]$, where $\epsilon \in L(R)$ means that `x` and `y` are may aliases. The language $L(R) - \{\epsilon\}$ represents other possible paths from the cell pointed to by `x` to the cell pointed to by `y`, but it is also possible that in some of the stores that the path matrix represents, there is no path at all from the cell pointed to by `x` to the cell pointed to by `y`.

Note that a path matrix represents a smaller set of stores if the language for a "D" entry is made smaller, and also if the language for a "P" entry is made smaller (see (3) and (4)).

Example 32 The following path matrix expresses a loop-invariant for the loop of `elem_reverse`:

pm	x	y	z
x	$\langle \epsilon, D\rangle$	$\langle \emptyset, P\rangle$	$\langle \emptyset, P\rangle$
y	$\langle \emptyset, P\rangle$	$\langle \epsilon, D\rangle$	$\langle \texttt{cdr}, D\rangle$
z	$\langle \emptyset, P\rangle$	$\langle \emptyset, P\rangle$	$\langle \epsilon, D\rangle$

(6)

The fact that $pm[\texttt{y},\texttt{z}]$ is $\langle \texttt{cdr}, D\rangle$ signifies that $\texttt{y} \rightarrow \texttt{cdr}$ must point to the cell that `z` points to.

Example 33 The following path matrix expresses a loop-invariant for the loop of `elem_delete`:

pm	prev	elem	c
prev	$\langle \epsilon, D\rangle$	$\langle \texttt{cdr}, P\rangle$	$\langle \emptyset, P\rangle$
elem	$\langle \emptyset, P\rangle$	$\langle \epsilon, D\rangle$	$\langle \emptyset, P\rangle$
c	$\langle \texttt{cdr}^*, P\rangle$	$\langle \epsilon \vert \texttt{cdr}^*.\texttt{prev}?.\texttt{cdr}, P\rangle$	$\langle \epsilon, D\rangle$

(7)

The fact that $pm[\texttt{prev},\texttt{elem}]$ is $\langle \texttt{cdr}, P\rangle$ signifies that $\texttt{prev} \rightarrow \texttt{cdr}$ may point to the cell pointed to by `elem`, but may also point to a cell that `elem` does not point to; in fact, the latter is the case at the beginning of the first loop iteration. Similarly, the fact that $pm[\texttt{c},\texttt{prev}]$ is $\langle \texttt{cdr}^*, P\rangle$ signifies that `prev` may be reachable from `c`. The fact that $pm[\texttt{c},\texttt{elem}]$ entry is $\langle \epsilon|\texttt{cdr}^*.\texttt{prev}?.\texttt{cdr}, P\rangle$ signifies that either `c` and `elem` point to the same cell, or else that as we traverse the list pointed to by `c`, we first reach a cell pointed to by `prev` and then the cell pointed to by `elem`.

Remark. The routing expressions that we allow in path matrices are more general than the ones allowed in [9, 11] in the following way:

– We allow arbitrary alternations and not just `car`|`cdr`.
– We follow [13] in allowing pointer-variable interrogations (e.g., `prev`, $\neg$`prev`) in routing expressions. This comes in handy in cases where several paths depend on each other (cf. the $pm[\texttt{c},\texttt{elem}]$ entry in path matrix (7)).

(The use of a less-general language of routing expressions in [9, 11] was motivated by the need to be able to compute efficiently a safe approximation to the path matrix at every program point.)

Since path matrices are an intuitive notation, we will not spend the space in directly formalizing the meaning of path matrices in terms of sets of stores. Instead, we now define the meaning of a path matrix by a formula in our language that characterizes the set of stores that a path matrix represents.

Definition 34 *For a regular expression R, let $\overline{R}$ denote the complement of R, i.e., a regular expression such that $L(\overline{R}) = \overline{L(R)} = \Sigma^* - L(R)$. For a path matrix pm, we define the formula φ_{pm} as follows:*

$$\varphi_{pm} \stackrel{\text{def}}{=} \bigwedge_{\mathtt{x},\mathtt{y} \in PVar, \langle R, D\rangle \in pm[\mathtt{x},\mathtt{y}]} \mathtt{x}\langle R\rangle\mathtt{y} \wedge \bigwedge_{\mathtt{x},\mathtt{y} \in PVar, \langle R, P\rangle \in pm[\mathtt{x},\mathtt{y}]} \neg\mathtt{x}\langle \overline{R}\rangle\mathtt{y} \tag{8}$$

This definition is justified by the discussion that follows Definition 31.

Example 35 Path matrix (6), which expresses a loop-invariant for the loop of `elem_reverse` (see Example 32), corresponds to the following formula:

$$\begin{array}{rrr} \mathtt{x}\langle\epsilon\rangle\mathtt{x} & \wedge\ \neg\mathtt{x}\langle\Sigma^*\rangle\mathtt{y} & \wedge\ \neg\mathtt{x}\langle\Sigma^*\rangle\mathtt{z} \\ \wedge\ \neg\mathtt{y}\langle\Sigma^*\rangle\mathtt{x} & \wedge\quad \mathtt{y}\langle\epsilon\rangle\mathtt{y} & \wedge\ \ \mathtt{y}\langle\mathtt{cdr}\rangle\mathtt{z} \\ \wedge\ \neg\mathtt{z}\langle\Sigma^*\rangle\mathtt{x} & \wedge\ \neg\mathtt{z}\langle\Sigma^*\rangle\mathtt{y} & \wedge\quad \mathtt{z}\langle\epsilon\rangle\mathtt{z} \end{array} \tag{9}$$

Formula (9) is less informative than the loop-invariant given as Formula (1) of Example 22. For example, with Formula (9) it is not known that `x` points to a list, because cyclic stores of the form shown in Figure 2 also satisfy (9).

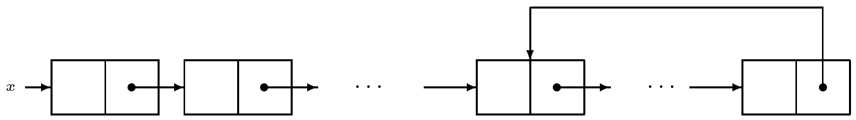

Fig. 2. A store with a shared node.

Example 36 Path matrix (10), which expresses a loop-invariant for the loop of `elem_delete` (see Example 33), corresponds to the following formula:

$$\begin{array}{rrr} \mathtt{prev}\langle\epsilon\rangle\mathtt{prev}\ \wedge & \neg\mathtt{prev}\langle\overline{\mathtt{cdr}}\rangle\mathtt{elem}\ \wedge & \neg\mathtt{prev}\langle\Sigma^*\rangle\mathtt{c} \\ \wedge\ \neg\mathtt{elem}\langle\Sigma^*\rangle\mathtt{prev}\ \wedge & \mathtt{elem}\langle\epsilon\rangle\mathtt{elem}\ \wedge & \mathtt{elem}\langle\Sigma^*\rangle\mathtt{c} \\ \wedge\quad \neg\mathtt{c}\langle\overline{\mathtt{cdr}^*}\rangle\mathtt{prev}\ \wedge & \neg\mathtt{c}\langle\overline{\epsilon|\mathtt{cdr}^*.\mathtt{prev}?.\mathtt{cdr}}\rangle\mathtt{elem}\ \wedge & \mathtt{c}\langle\epsilon\rangle\mathtt{c} \end{array} \tag{10}$$

Formula (10) is less informative than the loop-invariant given as Formula (2) of Example 23. In contrast to Formula (2), Formula (10) cannot be used to conclude that the use of `free` in `elem_delete` is correct; i.e., we cannot conclude that the location released by the statement `free(elem)` cannot be pointed to by `c`.

4 Representing Shape Graphs via Formulae

In this section, we study a structure-description formalism called *static shape graphs*, which, in addition to reachability information, allow certain "topological" properties of stores to be represented. There are many ways to define static shape graphs. For simplicity, we only consider the variant of static shape graphs defined in [21]. In Section 4.1, we give a formal definition of static shape graphs. Then, in Section 4.2, we construct a formula in L_r that exactly characterizes the set of stores represented by a static shape graph.

4.1 Static Shape Graphs

Below, we formally define static shape graphs. Unlike the stores defined in Section 2, static shape graphs are of an *a priori* bounded size, i.e., the number of shape nodes depends only of the size of the program being analyzed. This is needed by shape-analysis algorithms so that an iterative shape-analysis algorithm that computes static shape graphs for each program point will terminate.

Definition 41 *A* **static-shape-graph** *(SSG) is a finite directed graph that consists of two kinds of nodes —* **variables** *(i.e., $PVar$) and* **shape-nodes** *— and two kinds of edges —* **variable-edges** *and* **selector-edges**. *A shape-graph is represented by a quadruple $\langle shapeNodes, E_v, E_s, is\rangle$, where:*

- *$shapeNodes$ is a finite set of shape nodes. Every shape node $n \in ShapeNodes$ has the form $n = n_X$ where $X \subseteq PVar$. Such a node describes the cells that are simultaneously pointed to by all the pointer variables in X.*
  Graphically, we denote shape nodes by circles. The node $n_\emptyset$ is the "summary-node" since it represents all the cells that are not directly pointed to by any pointer variable, and therefore it is represented by a dotted circle.
- *E_v is the graph's set of variable-edges, each of which is denoted by a pair of the form $[\mathtt{x}, n_X]$, where $\mathtt{x} \in PVar$ and $n_X \in shapeNodes$. We assume that for every $\mathtt{x} \in PVar$, at most one variable-edge $[\mathtt{x}, n_X] \in E_v$ exists and $\mathtt{x} \in X$.*
  Graphically, we denote variable-edges by solid edges since they must exist.
- *E_s is the graph's set of selector-edges, each of which is denoted by a triple of the form $\langle n_X, sel, n_Y\rangle$, where $n_X, n_Y \in shapeNodes$ and $sel \in \{car, cdr\}$. We assume that for every $\mathtt{x} \in PVar$, $sel \in \{car, cdr\}$, and shape node n_X such that $[\mathtt{x}, n_X] \in E_v$, at most one selector-edge, $\langle n_X, sel, n_Y\rangle \in E_s$ exists. In contrast, there may be many selector-edges $\langle n_\emptyset, sel, n_Y\rangle \in E_s$ corresponding to different selector-edges emanating from cells represented by $n_\emptyset$.*
  Graphically, we denote selector-edges by dotted edges since they may or may not exist.
- *is (standing for "is shared") is a function of type $shapeNodes \to \{false, true\}$. It serves as a constraint to restrict the set of stores represented by a shape graph. When $n_\emptyset$ has more than one incoming selector edge and yet $is(n_\emptyset) = false$, we know that, for any memory cell c represented by $n_\emptyset$, at most one of the concrete representatives of these selector-edges can be an incoming edge of c.*

Graphically, we denote the fact that n_X is a shared node by putting "$is(n_X)$" inside the circle.

Example 42 The SSG that represents the store shown in Figure 2 is shown in Figure 3.

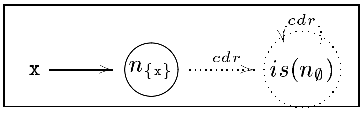

Fig. 3. The SSG that corresponds to the store shown in Figure 2.

4.2 From a Static Shape Graph to an L_rFormula

We are now ready to show how to construct the formula that captures the meaning of a static shape graph.

Definition 43 *Let $SG = \langle shapeNodes, E_v, E_s, is \rangle$ be an SSG. We define the graph $\widehat{SG}$ to be a directed graph, $\widehat{SG} = (N, A, l)$, with edges labeled by letters in Σ, where:*

- *N contains two nodes $u.in$ and $u.out$ for every shape node u in $shapeNodes$.*
- *$A \subseteq N \times N$ contains the following two types of labeled edges:*
  - *For every shape node n_X such that $[p_1, n_X], [p_2, n_X], \cdots, [p_n, n_X] \in E_v$ and $[p_{n+1}, n_X], [p_{n+2}, n_X], \cdots, [p_{n+k}, n_X] \notin E_v$, there is an edge $\langle n_X.in, n_Y.out \rangle$, labeled by $(p_1?.p_2?.\cdots.p_n?.\neg p_{n+1}?.\neg p_{n+2}?\cdots.\neg p_{n+k}?)$.*
  - *If there is a selector-edge $\langle n_X, sel, n_Y \rangle \in E_s$, there is an edge $a = (n_X.out, n_Y.in)$ from $n_X.out$ into $N_Y.in$, labeled sel.*
- *$l\colon A \to \Sigma$ maps edges into their labels.*

For any two nodes $n_X, n_Y \in shapeNodes$, let $r_{n_X.in \to n_Y.out}$ be the regular path expression over Σ that describes paths in $\widehat{SG}$ from $n_X.in$ into $n_Y.out$ (which can be computed by well-known methods, e.g., [25, 24]). For a finite set of regular expressions $S = \{r_1, r_2, \ldots, r_n\}$, $Rsum_S$ denotes the regular expression $r_1|r_2|\cdots|r_n$. Finally, for a regular expression r, $\overline{r}$ is the regular expression over Σ that describes the non-existing words in r, i.e., $L(\overline{r}) = \Sigma^* - L(r)$. Let us define the following formulae to characterize the different aspects of SG

$$
\begin{aligned}
\Phi^+_{val} &= \textstyle\bigwedge_{\mathtt{x} \in PVar, [\mathtt{x}, n_X] \in E_v} al(\mathtt{x}) \qquad \Phi^-_{val} = \textstyle\bigwedge_{\mathtt{x} \in PVar, [\mathtt{x}, n_X] \notin E_v} \neg al(\mathtt{x}) \\
\Phi^+_{veq} &= \textstyle\bigwedge_{\mathtt{x},\mathtt{y} \in PVar, [\mathtt{x}, n_X], [\mathtt{y}, n_X] \in E_v} \mathtt{x} = \mathtt{y} \; \Phi^-_{veq} = \textstyle\bigwedge_{\mathtt{x},\mathtt{y} \in PVar, [\mathtt{x}, n_X] \in E_v, [\mathtt{y}, n_X] \notin E_v} \mathtt{x} \neq \mathtt{y} \\
\Phi^-_{pal} &= \textstyle\bigwedge_{\mathtt{x} \in PVar, [\mathtt{x}, n_X] \in E_v} \neg al(\mathtt{x}\langle \overline{Rsum_{n_Y \in shapeNodes} r_{n_X.in \to n_Y.out}} \rangle) \\
\Phi^-_{r} &= \textstyle\bigwedge_{[\mathtt{x}, n_X], [\mathtt{y}, n_Y] \in E_v} \neg \mathtt{x}\langle \overline{r_{n_X.in \to n_Y.out}} \rangle \mathtt{y} \\
\Phi^-_{hs} &= \textstyle\bigwedge_{\mathtt{x} \in PVar, [\mathtt{x}, n_X] \in E_v} \neg hs(\mathtt{x}\langle \overline{Rsum_{n_Y \in shapeNodes, is(v)=1} r_{n_X.in \to n_Y.out}} \rangle)
\end{aligned}
$$

Finally, the formula $\Phi[SG]$ is the conjunction of these formulae.

Lemma 44 *For every store S and SSG SG, S is represented by SG if and only if $[\![\Phi[SG]]\!]^S$ is true.*

5 Extracting Information from Program Analyses via L_r Formulae

Many interesting properties of linked data structures can be expressed as L_r formulae:

- For example, the formula $\Psi \stackrel{\text{def}}{=} (x = x.cdr)$, expresses the property "x points to a cell that has a self-cycle". This information can be used by an optimizing compiler to determine whether it is profitable to generate a pre-fetch for the next element [18].
- It is possible to express in L_r that two pointer-access paths point to different memory cells (i.e., they are not may-aliases), which is important both for optimization and in tools for aiding software understanding.
- The reachability and sharing predicates can also be useful, for example, to improve the performance of garbage-collection algorithms and to parallelize programs.

In principle, L_r provides a uniform basis for using the results of analyses that yield either path matrices or static shape graphs in program optimizers and in tools for aiding software understanding. For instance, Figure 4 shows one of the SSGs SG that arises at the loop header in an analysis of `elem_reverse`. It can be shown that Ψ is not satisfiable by any store that is represented by SG. This means that x does not point to a cell that has a self-cycle in any of the stores that SG represents. This can be determined automatically with our approach by showing that $\Phi[SG] \wedge \Psi$ is not satisfiable. Similarly, by translating a path matrix M (obtained from a path-matrix-based program-analysis algorithm) into the corresponding L_r formula $\Phi[M]$ and checking whether $\Phi[M] \wedge \Psi$ is satisfiable, one can verify automatically whether x could point to a cell that has a self-cycle in any of the stores represented by M.

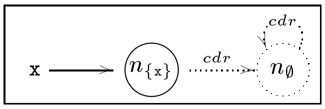

Fig. 4. An SSG, SG, that represents acyclic lists of length two or more that are pointed to by variable x.

6 The Decidability of L_r

Theorem 61 *L_r is decidable.*
Sketch of Proof: Prior to directly approaching the question of decidability of L_r, one first proves a normalization lemma showing that the routing expressions mentioned in formulae can be rewritten in such a way that they deal only with paths that avoid all nodes pointed by pointer expressions that are mentioned in the formula (i.e. pointer expressions that occur in some constraint or program variables that occur in some pointer-variable interrogation). That is, they assert only reachability of shared nodes or pointer expressions via paths that traverse nodes in the heap. One proves this normalization lemma by breaking down path expressions that may cross mentioned pointer expressions into component path expressions that do not cross mentioned pointer expressions.

The decidability of logic L_r follows from showing that that L_r has the *bounded model property*: that is, there is a computable numerical function f such that any sentence ϕ of L_r that is consistent has a model of size bounded by $f(|\phi|)$. This technique is one of the most common in logical decision procedures [2]. It immediately shows the existence of a crude decision procedure: one enumerates all possible stores of size $f(|\phi|)$ searching for a model. (Note that the approach sketched here is intended only to give a comprehensible demonstration of decidability, not to give a practical decision procedure.) The proof of the bounded model property proceeds by starting with an arbitrary concrete store G satisfying a formula ϕ and showing that G can be diminished to a model of size $|f(|\phi|)$ (for a particular f given in the proof) while preserving all atomic formulae in ϕ.

The normalization theorem above implies that in this shrinking process one only has to preserve properties that deal with paths through the heap (reachability, heap-sharing, etc.) and equalities and inequalities between a fixed set of pointer expressions. This shrinking is then done in three phases: first, the original store G is "pruned" to get a model that is a union of trees: in the process, some information about the sharing of nodes is lost, but extra labels are added to the nodes to maintain this information. These "auxiliary labels" indicate that certain nodes in the tree correspond to nodes associated with a particular pointer expression in the original store, and that certain nodes in the tree were shared in the original store.

We then make use of classical decidability results on reachability expressions on finite trees ([26], summarized also in [2]) to shrink each of these trees to smaller trees that satisfy the same properties as the union of trees produced in stage one. The "properties" mentioned here are obtained by taking the original reachability, heap-sharing, and allocation constraints and transforming them to expressions in monadic second-order logic that express how to reach the auxiliary labels mentioned above.

Finally, the shrunken set of trees are glued together to restore sharing information lost in the first phase: multiple nodes that have been annotated as associated with the same pointer expression are identified, and nodes that were annotated as being shared heap nodes are made into shared nodes. The normalization results are used in a crucial way in this glueing stage, since the glueing can create many new paths within the represented store. Glueing cannot, however, create new paths through the heap in any store, since the glueing process only identifies nodes associated with pointer expressions mentioned in the formula (or in unshared paths leading to such nodes). Since normalization implies that we are only concerned with preserving the existence and nonexistence of paths that lie strictly within the heap, this is sufficient.

Figures 5 and 6 show how the proof might work for the formula Φ:

$$\Phi \stackrel{\text{def}}{=} x\langle \texttt{car}^* \rangle y \wedge x\langle (\texttt{cdr.cdr})^* \rangle y \wedge \neg x\langle (\texttt{cdr.cdr.cdr})^* \rangle y \wedge y\langle \texttt{car}^* \rangle z \wedge y\langle (\texttt{cdr.cdr.cdr})^* \rangle z.$$

We start with a store in Figure 5 that satisfies Φ, and then prune it into a set of trees. The auxiliary labels y' and y'' keep track of the fact that these nodes in the tree must at some point be pointed to by y. In Figure 6, the trees are decreased in size, while preserving analogs of the reachability statements: e.g., the node labeled y can reach a copy of the node z with a $(\texttt{cdr.cdr.cdr})^*$ path, and x cannot reach a copy of y with a $(\texttt{cdr.cdr.cdr})^*$ path. In the final stage, the tree-like model is glued together to form a traditional store that satisfies Φ.

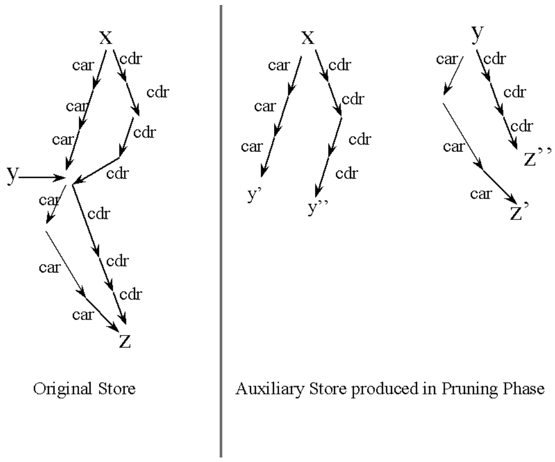

Fig. 5. The pruning stage of the proof

7 Related Work

Jensen et al. have also defined a decidable logic for describing properties of linked data structures [13]. It is interesting to compare the two approaches:

- The logic of Jensen et al. allows quantifications on pointer expressions, which is forbidden in L_r. Instead, L_r allows stating both sharing constraints and

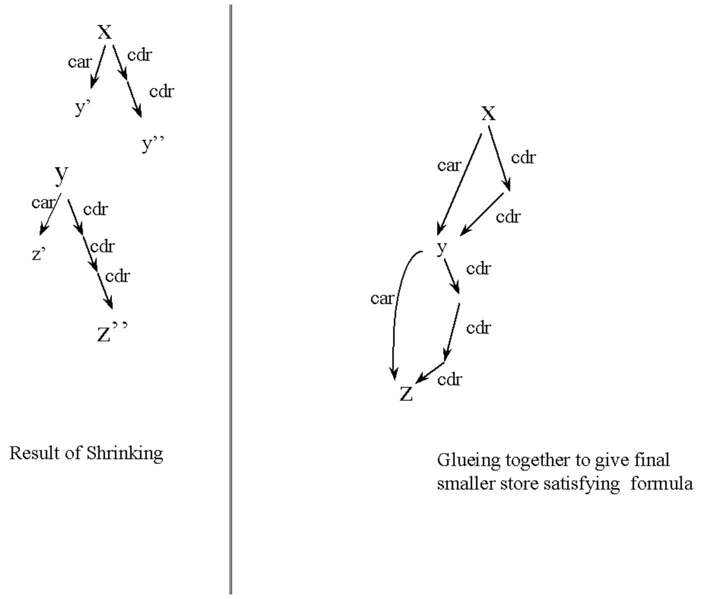

Fig. 6. The shrinking and glueing stages of the proof

allocation constraints. However, both of these can be encoded using their logic:
  - Sharing constraints that can be encoded using quantifications.
  - Allocation constraints can be encoded using tests for `NULL` in routing expressions.
- L_r imposes a limitation on routing expressions by forbidding testing for `NULL` and for garbage cells.
- On the other hand, L_r generalizes the logic of Jensen et al. in the following ways:
  - L_r allows multiple selectors, which enables L_r formulae to describe properties of general directed graphs as opposed to just lists.[1]
  - The reachability constraints in L_r formulae allow one to test simultaneous pointer inequities, which is crucial for capturing the strength of the variant of static shape graphs defined in [21].

In summary, the formulae of Jensen et al. are more expressive than L_r formulae, but they can only state properties of lists and trees, whereas L_r can state properties of arbitrary graph data structures.

Klarlund and Schwartzbach defined a language for defining *graph types*, which are tree data structures with non-tree links defined by auxiliary tree-path expressions [16]. In the application they envision, a programmer would be able to declare variables of a given graph type, and write code to mutate the "tree

[1] [13] sketches an extension of their technique to trees, which involves multiple selectors, but they do not handle general directed graphs.

backbone" of these structures. After a mutation operation, the runtime system would automatically apply an update operation that they define, which updates the non-tree links. The graph-type definition language is unsuitable for describing arbitrary store graphs, and the fact that the update operations are limited does not allow the programmer to write arbitrary pieces of code. (However, the latter property is a significant advantage for the intended application — a programming language supporting controlled destructive updating of graph data structures.)

The ADDS formalism of Hendren et al. is an annotation language for expressing loop invariants and pre- and post-conditions of statements and procedures [10]. From a programmer's point of view, an advantage of a logic like L_r over ADDS is that L_r is strong enough to allow stating properties of the kind that arise at intermediate points of a procedure, when a data structure is in the process of being traversed or destructively updated. For example, ADDS cannot be used to state the loop invariant of Example 22 because the relationship between `x` and `y` cannot be expressed. Because it is lacking in expressive power, ADDS is mainly useful as a documentation notation for type definitions, function arguments, and function return values. Hendren et al. propose to handle this limitation of ADDS by extending it with the ability to use a certain limited class of reachability properties between variables (of the kind used in the path matrices defined in [9]).

L_r goes beyond ADDS in the following ways:

- L_r permits stating properties of cyclic data structures.
- The routing expressions used in L_r formulae are general regular expressions (with pointer-variable interrogations).
- L_r is closed under both conjunction and negation. In contrast, ADDS cannot express the loop invariant in Example 23 because of the implication.

It should be noted that currently the notion of heap sharing in L_r is weaker than the ADDS notion of "dimension". It is easy to generalize L_r to include this concept without affecting its decidability. We did not do so in this paper because we wanted to stay with two selectors.

Finally, it should be noted that both L_r and ADDS do not allow stating connectivity properties of the form $\mathtt{x}\langle R_1 \rangle = \mathtt{y}\langle R_2 \rangle$. We believe that L_r can be generalized to handle this. (A limited form of such connectivity properties, restricted to be in the form $\mathtt{x}\langle(\mathtt{car}|\mathtt{cdr})^*\rangle = \mathtt{y}\langle(\mathtt{car}|\mathtt{cdr})^*\rangle$, was proposed in [8, 7].)

L_r is incomparable to Deutsch's symbolic aliases [4, 5]: Symbolic aliases allow the use of full-blown arithmetic, which cannot be used in a decidable logic. On the other hand, symbolic-alias expressions are not closed under negation. For instance, there is no way to express must-alias relationships using symbolic aliases. Thus, the loop invariant used in Example 23 cannot be expressed with symbolic aliases.

In [6], Fradet and Le Métayer use graph grammars to express interesting properties of the data structures of a C-like language. Graph grammars can be a more natural formalism than logic for describing certain topological properties of stores. However, graph grammars are not closed under intersection and negation, and problems such as the inclusion problem are not decidable. In terms of expressive power, the structure-description formalism of [6] is incomparable to the one proposed in the present paper.

It should be noted that the approach given here is limited in several ways: The approach we have taken is to develop decidable, logic-based languages for capturing topological properties of a broad class of linked data structures. Undecidability results in predicate logic give many hard limitations on the expressiveness of such languages: For example, no such language exists that is closed under first-order quantification and boolean connectives. Although logic-based formalisms can be more succinct in expressing properties of linked data structures, they can also be more verbose; in particular, the output from our translation algorithms can be significantly more verbose than the input. For example, with the translation from a static shape graph SG into L_r formula $\Phi[SG]$ given in Section 4.2, the size of $\Phi[SG]$ can be exponential in $|SG|$.

There are a few properties that cannot be expressed in L_r, including: (i) whether a store contains a garbage cell (i.e., a cell not accessible from any variable), and (ii) whether a tree is balanced (or almost balanced, such as the condition used in AVL trees). It may be difficult to extend L_r to handle these sorts of properties. However, such properties go well beyond the scope of current optimizing compilers and tools for aiding software understanding.

Acknowledgements

This work was supported in part by the NSF under grants CCR-9625667 and CCR-9619219, by the United States-Israel Binational Science Foundation under grant 96-00337, and by a Vilas Associate Award from the Univ. of Wisconsin.

References

1. U. Assmann and M. Weinhardt. Interprocedural heap analysis for parallelizing imperative programs. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Programming Models For Massively Parallel Computers*, pages 74–82, Washington, DC, September 1993. IEEE Press.
2. E. Boerger, E. Graedel, and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1996.
3. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.
4. A. Deutsch. A storeless model for aliasing and its abstractions using finite representations of right-regular equivalence relations. In *IEEE International Conference on Computer Languages*, pages 2–13, Washington, DC, 1992. IEEE Press.
5. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 230–241, New York, NY, 1994. ACM Press.
6. Pascal Fradet and Daniel Le Metayer. Shape types. In *Symp. on Princ. of Prog. Lang.*, New York, NY, 1997. ACM Press.
7. R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Symp. on Princ. of Prog. Lang.*, New York, NY, 1998. ACM Press.
8. R. Ghiya and L.J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proc. of the 8th Int. Workshop on Lang. and Comp. for Par. Comp.*, number 1033 in Lec. Notes in Comp. Sci., pages 515–534, Columbus, Ohio, August 1995. Springer-Verlag.

9. L. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.
10. L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 249–260, New York, NY, June 1992. ACM Press.
11. L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Par. and Dist. Syst.*, 1(1):35–47, January 1990.
12. S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 28–40, New York, NY, 1989. ACM Press.
13. J.L. Jensen, M.E. Joergensen, N.Klarlund, and M.I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 1997.
14. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.
15. N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Symp. on Princ. of Prog. Lang.*, pages 66–74, New York, NY, 1982. ACM Press.
16. N. Klarlund and M. Schwartzbach. Graph types. In *Symp. on Princ. of Prog. Lang.*, New York, NY, January 1993. ACM Press.
17. J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 21–34, New York, NY, 1988. ACM Press.
18. C.-K. Luk and T.C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.
19. J. Plevyak, A.A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lec. Notes in Comp. Sci.*, pages 37–57, Portland, OR, August 1993. Springer-Verlag.
20. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. Tech. Rep. TR-1383, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, July 1998. Available at "http://www.cs.wisc.edu/wpis/papers/parametric.ps".
21. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50, January 1998.
22. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 1999. Available at "http://www.cs.wisc.edu/wpis/papers/popl99.ps".
23. J. Stransky. A lattice for abstract interpretation of dynamic (Lisp-like) structures. *Inf. and Comp.*, 101(1):70–102, Nov. 1992.
24. R.E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.
25. R.E. Tarjan. A unified approach to path problems. *J. ACM*, 28(3):577–593, 1981.
26. J.W. Thatcher and J.B. Wright. Generalized finite automata with an application to a decision problem of second order logic. *Math. Syst. Theory*, 2:57–82, 1968.
27. E. Y.-B. Wang. *Analysis of Recursive Types in an Imperative Language.* PhD thesis, Univ. of Calif., Berkeley, CA, 1994.