The VeriFast Program Verifier

Bart Jacobs * Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{bart.jacobs,frank.piessens}@cs.kuleuven.be

Abstract

This note describes a separation-logic-based approach for the specification and verification of safety properties of pointer-manipulating imperative programs. We describe the approach for the C language.

The safety properties to be verified are specified as annotations in the source code, in the form of function preconditions and postconditions expressed as separation logic assertions. To enable rich specifications, the user may include additional annotations that define inductive datatypes, primitive recursive pure functions over these datatypes, and abstract predicates (i.e. named, parameterized assertions). A restricted form of existential quantification is supported in assertions in the form of pattern matching.

Verification is based on forward symbolic execution, where memory is represented as a separate conjunction of points-to assertions and abstract predicate assertions, and data values are represented as first-order logic terms with a set of constraints. Abstract predicates must be folded and unfolded explicitly using ghost statements. Rewritings of the abstract state that require induction, or derivations of facts over data values that require induction, can be done by defining lemma functions, which are like ordinary C functions except that it is checked that they terminate. Specifically, when a lemma function performs a recursive call, either the recursive call must apply to a strict subset of memory, or one of its parameters must be an inductive value whose size decreases at each recursive call.

Assertions over data values are delegated to an SMT solver, formulated as queries against an axiomatization of the inductive datatypes and recursive pure functions. Importantly, no exhaustiveness axioms are included in this axiomatization; this prevents the SMT solver from performing case analysis on inductive values. Combined with a measure to prevent infinite reductions due to *self-feeding recursions*, this ensures termination of the SMT solver.

The time complexity of verification is unbounded in theory. Specifically, since recursive pure functions of arbitrary time complexity may be defined, there is no bound on the time complexity of SMT queries. Furthermore, the approach, as currently implemented, does not perform joining of symbolic execution paths after conditional constructs; therefore, the number of symbolic execution steps is exponential in the number of such constructs. However, since no significant search is performed implicitly by the verifier or the SMT solver, performance is very good in practice.

A prototype implementation and example annotated programs are available at http://www.cs.kuleuven.be/~bartj/verifast/.

1. Introduction

We introduce the approach through an example annotated C program that implements a linked list ADT. Successive figures show successive fragments of the example program.

* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

```
struct node {
  struct node *next;
  int value;
};

predicate node(struct node *n,
    struct node *next, int value)
  requires n→next ↦ next * n→value ↦ value
    * malloc_block_node(n);

struct node *create_node(struct node *next, int value)
  requires emp;
  ensures node(result, next, value);
{
  struct node *n := malloc(sizeof(struct node));
  n→next := next;
  n→value := value;
  close node(n, next, value);
  return n;
}
```

Figure 1. Example demonstrating abstract predicates and ghost statements (Note: annotations are shown on a gray background. Also, for readability, we typeset some operators differently from the implementation.)

```
inductive list = nil | cons(int, list);

predicate lseg(struct node *n1, struct node *n2, list v)
  requires n1 = n2 ? v = nil
    : node(n1, ?n, ?h) * lseg(n, n2, ?t) * v = cons(h, t);
```

Figure 2. Example demonstrating inductive datatype definitions, recursive abstract predicates, conditional assertions, and pattern matching

Figure 1 shows function *createNode*. It uses an abstract predicate to hide the internal layout of a node. The **close** ghost statement removes the points-to assertions for the individual fields of n from the abstract memory representation, and adds a *node* abstract predicate assertion, as expected by the postcondition.

Figure 2 shows a way to denote a piece of memory containing a set of consecutive nodes. Specifically, abstract predicate $lseg(n1, n2, v)$ represents a set of consecutive nodes where the first node is at $n1$ and the last node's next pointer points to $n2$, and

the nodes store the list of integers v. As a special case, if $n1$ equals $n2$, the predicate denotes the empty piece of memory.

During symbolic execution of a function, assertions are *produced* and *consumed*. Producing a points-to assertion or an abstract predicate assertion means adding it to the abstract memory, and consuming it means removing a matching assertion from the abstract memory. If no matching assertion is present in the abstract memory, an error is reported.[1] If the assertion being consumed contains patterns, the matching process binds the pattern variables; their scope includes the rest of the assertion, or if the pattern occurs in a function body or precondition, its scope includes the rest of the function. Producing a pure assertion (i.e., a boolean expression), means adding it to the set of constraints, and consuming it means asking the SMT solver to check that it follows from the current set of constraints. Producing or consuming a separate conjunction means first producing, resp. consuming the first operand, and then producing, resp. consuming the second operand. Producing or consuming **emp** does nothing. If during execution of a function, a conditional construct is encountered, then the remainder of the execution is performed once for each branch of the construct, after adding the corresponding constraint to the constraint set. The conditional constructs include the if and switch statements, the if-then-else assertions, and the switch assertions.

Execution of a function starts with an empty memory and an empty set of data value constraints. Then, the precondition is produced. Then, each statement is executed. And finally, the postcondition is consumed. If subsequently, any assertions are left in the abstract memory, this is considered a potential memory leak and an error is reported. Execution of a function call statement proceeds by first consuming the call's precondition and then producing its postcondition. Execution of an open ghost statement proceeds by first consuming the abstract predicate assertion and then producing its body. Execution of a close ghost statement proceeds by first consuming the predicate's body and then producing the abstract predicate assertion. Patterns may be used as abstract predicate arguments in an open statement, but in the current implementation they cannot be used as arguments in a close statement.

Figure 3 shows the first part of the client-visible interface of the linked list ADT. The implementation keeps a sentinel node at the end of the list, and it keeps a pointer to the first node and to this sentinel node. The dummy patterns (_) in the definition of the *llist* predicate indicate that the *next* and *value* fields of the sentinel node are insignificant.

Figure 4 shows a *lemma function*, which is like a C function except that it is declared in an annotation and the verifier checks that it terminates and that has no effect on memory (i.e. it does not allocate, free, or write to memory). The only effect of calling a lemma function is that it rewrites the abstract memory representation into a semantically equivalent but syntactically different one, and/or that it adds constraints to the set of data value constraints.

In this example, there is no net change to the memory representation; all the lemma function does is add a constraint. Specifically, given two nodes for which there are separate abstract predicate assertions in the memory representation, the lemma produces a constraint that says that the nodes are distinct.

Such distinctness constraints are not produced automatically by the verifier for abstract predicate assertions, since the fact that two abstract predicate assertions calling the same abstract predicate appear in memory does not imply anything about distinctness of the arguments. However, the verifier produces them for points-to assertions. Specifically, when producing a points-to assertion $t{\rightarrow}f \mapsto v$, then for any existing points-to assertion $t'{\rightarrow}f \mapsto v'$ in the memory representation, a constraint $t \neq t'$ is added automatically. In the example, this occurs during execution of the second open statement.

[1] In the current implementation, if multiple matching assertions are present, an "ambiguous match" error is reported. This may change in the future.

```
struct llist {
  struct node *first;
  struct node *last;
};

predicate llist(struct llist *l, list v)
  requires l→first ↦ ?fn * l→last ↦ ?ln * lseg(fn, ln, v)
    * node(ln, _, _) * malloc_block_llist(l);

struct llist *create_llist()
  requires emp;
  ensures llist(result, nil);
{
  struct llist *l := malloc(sizeof(struct llist));
  struct node *n := create_node(0, 0);
  l→first := n;
  l→last := n;
  close lseg(n, n, nil);
  close llist(l, nil);
  return l;
}
```

Figure 3. Example demonstrating dummy patterns

```
lemma void distinct_nodes(
    struct node *n1, struct node *n2)
  requires node(n1, ?n1n, ?n1v) * node(n2, ?n2n, ?n2v);
  ensures node(n1, n1n, n1v) * node(n2, n2n, n2v)
    * n1 ≠ n2;
{
  open node(n1, _, _);
  open node(n2, _, _);
  close node(n1, n1n, n1v);
  close node(n2, n2n, n2v);
}
```

Figure 4. Example demonstrating lemma functions, distinctness constraint production and patterns in open statements

Figure 5 shows the second client-visible list ADT function, function *add*. It adds a value to the end of the list. Its contract describes its effect on the ADT's abstract value using the *fixpoint function add*. (Note that fixpoint function names and non-fixpoint (i.e., regular or lemma) function names are in separate namespaces; the former may occur only in expressions in annotations, whereas the latter may occur only in call statements.)

A fixpoint function is not allowed to read or modify memory. Its body must be a switch statement over one of the function's parameters. We call this parameter the function's *inductive parameter*. The body of each clause of the switch statement must be a return statement. A fixpoint function may call other fixpoint functions, but not regular functions or lemma functions. Furthermore, to ensure termination, any call must either be a call of a fixpoint function declared earlier in the program, or it must be a direct recursive call where the argument for the inductive parameter is a variable bound by the switch statement.

```
fixpoint list add(list v, int x) {
  switch (v) {
    case nil : return cons(x, nil);
    case cons(h, t) : return cons(h, add(t, x));
  }
}

lemma add_lemma(struct node *n1, struct node *n2,
    struct node *n3)
  requires lseg(n1, n2, ?v) * node(n2, n3, ?x)
    * node(n3, _, _);
  ensures lseg(n1, n3, add(v, x)) * node(n3, _, _);
{
  distinct_nodes(n2, n3);
  open lseg(n1, _, _);
  if (n1 = n2) {
    close lseg(n3, n3, nil);
  } else {
    distinct_nodes(n1, n3);
    open node(n1, ?n1n, ?n1v);
    add_lemma(n1n, n2, n3);
    close node(n1, n1n, n1v);
  }
  close lseg(n1, n3, add(v, x));
}

void add(struct llist *l, int x)
  requires llist(l, ?v);
  ensures llist(l, add(v, x));
{
  open llist(l, v);
  struct node *n := create_node(0, 0);
  struct node *nl := l→last;
  open node(nl, _, _);
  nl→next := n;
  nl→value := x;
  close node(nl, n, x);
  l→last := n;
  struct node *nf := l→first;
  add_lemma(nf, nl, n);
  close llist(l, add(v, x));
}
```

Figure 5. Example demonstrating fixpoint functions and recursive lemma functions

```
int removeFirst(struct llist *l)
  requires llist(l, ?v) * v ≠ nil;
  ensures llist(l, ?t) * v = cons(result, t);
{
  open llist(l, v);
  struct node *nf := l→first;
  open lseg(nf, ?nl, v);
  open node(nf, _, _);
  struct node *nfn := nf→next;
  int nfv := nf→value;
  free(nf);
  l→first := nfn;
  open lseg(nfn, nl, ?t);
  close lseg(nfn, nl, t);
  close llist(l, t);
  return nfv;
}
```

Figure 6. Example demonstrating execution splits due to conditional constructs in the bodies of predicates being opened

```
void dispose(struct llist *l)
  requires llist(l, _);
  ensures emp;
{
  open llist(l, _);
  struct node *n := l→first;
  struct node *nl := l→last;
  while (n ≠ nl)
    invariant lseg(n, nl, _);
  {
    open lseg(n, nl, _);
    open node(n, _, _);
    struct node *next := n→next;
    free(n);
    n := next;
  }
  open lseg(n, n, _);
  open node(l, _, _);
  free(nl);
  free(l);
}
```

Figure 7. Example demonstrating loops

Regular function *add* creates a new node to serve as the new sentinel node, then updates the old sentinel node's fields, and finally calls the lemma function *add_lemma* to merge the old sentinel node into the *lseg* abstract predicate assertion. The lemma function does so using recursion. Lemma functions may perform recursive calls, but only direct recursive calls, and termination is ensured by checking that at each recursive call either the size of the piece of memory that the function operates on decreases (specifically, after consuming the precondition there must be a points-to assertion left in the memory representation), or, similar to a fixpoint function, the function's body is a switch statement over one of its parameters, and the argument for the inductive parameter in the recursive call is bound by this switch statement. Note that a recursive lemma constitutes an inductive proof of the fact that the precondition implies the postcondition.

Figure 6 shows a function that removes the first element from a list. It requires that the list is non-empty. When the *lseg* starting at *nf* is opened, an execution split occurs because the body of this predicate is an if-then-else predicate. The verifier notices immediately that the then branch is infeasible and does not continue execution on this branch.

Notice also that the first node is freed. A statement *free*(p); looks for an assertion of the form *malloc_block_*$T(p)$, and then for points-to assertions of the form $p{\rightarrow}f \mapsto v$, for each field f of T, and it consumes all of these assertions.

```
void main()
  requires emp;
  ensures emp;
{
  struct llist *l := create_llist();
  add(l, 10);
  add(l, 20);
  add(l, 30);
  add(l, 40);
  int x0 := removeFirst(l);
  assert(x0 = 10);
  int x1 := removeFirst(l);
  assert(x1 = 20);
  dispose(l);
}
```

Figure 8. Example client program for the list ADT

Figure 7 shows function *dispose*, which takes a list of arbitrary length. It first frees all proper nodes, then it removes the remaining empty *lseg* assertion, then it frees the sentinel node, and finally it frees the **struct** *llist* object itself. A loop invariant must be provided for each loop. Execution of a loop proceeds by first consuming the loop invariant, and then proceeding execution along two branches: in one branch, fresh data value symbols are assigned to all locals modified by the loop, then the loop condition is produced, then the loop invariant is produced, then the loop body is executed, and finally the loop invariant is consumed. If any assertions remain, this is considered a leak error. In the other branch, first fresh data value symbols are assigned to all locals modified by the loop, then the negation of the loop condition is produced, then the loop invariant is produced, and then execution proceeds after the loop statement.

Figure 8 wraps up the example by showing an example client program for the list ADT. This program verifies; it follows that all assert statements succeed and no memory is leaked.

Notice that the SMT solver successfully evaluates the *add* fixpoint function applications.

2. Ensuring SMT solver termination

The verifier generates a set of axioms from the inductive datatype definitions and the fixpoint function definitions and uses an SMT solver to solve queries against this axiomatization. The axioms consist of the disjointness, injectiveness, and size axioms for the datatype constructors, and the reduction axioms for the fixpoint functions. Specifically, there is a reduction axiom for each fixpoint function f and for each constructor c of the type of its inductive parameter.

For example, the axioms generated for the example are:

$$
\begin{array}{l}
nil\_tag \neq cons\_tag \\
list\_tag(nil) = nil\_tag \\
\forall ht.list\_tag(cons(h,t)) = cons\_tag \\
\forall ht.cons\_proj0(cons(h,t)) = h \\
\forall ht.cons\_proj1(cons(h,t)) = t \\
size(nil) = 0 \\
\forall ht.size(cons(h,t)) = 1 + size(t) \\
\forall x.add(nil,x) = cons(x,nil) \\
\forall htx.add(cons(h,t),x) = cons(h,add(t,x))
\end{array}
$$

Note: for each quantifier, the left-hand side (and only the left-hand side) of the equation is used as the trigger.

In general, when given such an axiomatization, an SMT algorithm that consists of instantiating quantifiers arbitrarily until all quantifiers have been applied to all matching E-graph-nodes or an inconsistency is found, does not always terminate for all queries. For example, such an algorithm does not terminate for the below input file.

```
inductive nat = zero | succ(nat);

fixpoint nat foo(nat n, nat k) {
  switch (n) {
    case zero : return zero;
    case succ(n) : return succ(foo(n, succ(k)));
  }
}

fixpoint int nat2int(nat n) {
  switch (n) {
    case zero : return 0;
    case succ(n) : return 1 + nat2int(n);
  }
}

lemma void bar(nat n)
  requires n = foo(succ(n), zero);
  ensures nat2int(n) = 0;
{
}
```

The cause of the non-termination in this example is that each reduction of a *foo* node produces a new reducible *foo* node. Moreover, after the first reduction, each next inductive argument is a new node, generated by an earlier reduction. Therefore, the reduction chain in this example is a *self-feeding* reduction chain.

We propose a modification of the SMT algorithm which prevents self-feeding reduction chains, by keeping track of reduction chains and of the origin of each constructor node. A reduction is performed only if the constructor node that serves as the inductive argument was not generated by the reduction chain to which the fixpoint function node belongs. We believe this ensures termination, since a non-terminating search path must include an infinite number of reductions. Therefore, there must be an infinite reduction chain. This chain must be fed by an infinite path of constructor nodes. If we always exhaust the size axioms before applying reduction axioms, the path cannot be a cycle, since a cycle would be detected by the size axioms. Therefore, the path must include infinitely many distinct constructor nodes. These must have been generated by an infinite reduction chain; either the same one, or one whose fixpoint function was declared earlier in the program. By induction on the number of preceding fixpoint functions, one can obtain a contradiction.

Further detailing the algorithm and the proof is future work.

Note: We have not yet implemented this algorithm. The current implementation uses the Z3 SMT solver, which does not implement this algorithm. However, queries against our theory using Z3 are guaranteed to terminate, since Z3 limits the number of instantiations of any given quantifier. Since our theory does not contain nested quantifiers and therefore does not generate new quantifiers dynamically, this guarantees termination. However, this also limits the power of the algorithm. For example, whereas our proposed algorithm successfully fully reduces all ground terms, Z3 does not.

3. Performance

Since neither the verifier itself nor the SMT solver need to perform any significant amount of search, verification time is predictable and low. The table below shows indicative verification times for the two example programs that ship with the current release of the verifier.

Program	Nb. C stmts	Nb. ghost stmts	Verif. time
linkedlist.c	90	113	0.25s
composite.c	38	287	1.5s

4. Grammar

The current implementation supports only a limited subset of C. The grammar of the current implementation is as follows:

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{decl}^* \\
\mathit{decl} & ::= & \mathit{structdecl} \mid \mathit{inddecl} \mid \mathit{fpdecl} \\
 & & \mid \mathit{preddecl} \mid \mathit{fundecl} \\
\mathit{structdecl} & ::= & \mathbf{struct}\ s\ \{\ \mathit{fielddecl}^*\ \}\ ; \\
\mathit{fielddecl} & ::= & t\ f\ ; \\
t & ::= & \mathbf{struct}\ s * \mid \mathbf{int} \mid i \\
\mathit{inddecl} & ::= & \mathbf{inductive}\ i = \mathit{ctordecl}^*\ ; \\
\mathit{ctordecl} & ::= & \text{`|'}\ c(t^*) \\
\mathit{fpdecl} & ::= & \mathbf{fixpoint}\ t\ g(\mathit{paramdecl}^*) \\
 & & \{\ \mathbf{switch}\ (x)\ \{\ \mathit{fpclause}^*\ \}\ \} \\
\mathit{fpclause} & ::= & \mathbf{case}\ c(x^*) : \mathbf{return}\ e\ ; \\
\mathit{paramdecl} & ::= & t\ x \\
e & ::= & x \mid n \mid \neg e \mid e\ \mathit{binop}\ e \mid g(e^*) \mid e\ ?\ e\ :\ e \\
\mathit{binop} & ::= & = \mid \neq \mid < \mid \leq \mid + \mid - \mid \wedge \mid \vee \\
\mathit{preddecl} & ::= & \mathbf{predicate}\ p(\mathit{paramdecl}^*)\ \mathbf{requires}\ a; \\
a & ::= & \mathbf{emp} \mid e \mid e{\rightarrow}f \mapsto \mathit{pat} \mid p(\mathit{pat}^*) \mid a * a \\
 & & \mid e\ ?\ a\ :\ a \mid \mathbf{switch}\ (e)\ \{\ \mathit{aclause}^*\ \} \\
\mathit{pat} & ::= & e \mid ?x \mid \_ \\
\mathit{aclause} & ::= & \mathbf{case}\ c(x^*) : \mathbf{return}\ a\ ; \\
\mathit{fundecl} & ::= & \mathbf{lemma}^?\ \mathit{rettype}\ h(\mathit{paramdecl}^*) \\
 & & \mathbf{requires}\ a;\ \mathbf{ensures}\ a;\ \{\ s^*\ \} \\
\mathit{rettype} & ::= & \mathbf{void} \mid t \\
s & ::= & t\ x := h(e^*); \mid t\ x := e{\rightarrow}f; \mid t\ x := e; \mid x := e; \\
 & & \mid e{\rightarrow}f := e; \mid \mathbf{if}\ (e)\ \{\ s^*\ \}\ \mathbf{else}\ \{\ s^*\ \} \\
 & & \mid \mathbf{switch}\ (e)\ \{\ \mathit{sclause}^*\ \} \\
 & & \mid \mathbf{while}\ (e)\ \mathbf{invariant}\ a;\ \{\ s^*\ \} \\
\mathit{sclause} & ::= & \mathbf{case}\ c(x^*) : s^*
\end{array}
$$

5. Related work

Reynolds (2002) introduced separation logic. Smallfoot (Berdine et al. 2004, 2005, 2006) is a tool that performs symbolic execution using separation logic. This technique has been extended for greater automation (Yang et al. 2008), for termination proofs (Brotheston et al. 2008; Cook et al. 2007), for fine-grained concurrency (Calcagno et al. 2007), for lock-based concurreny (Gotsman et al. 2007), and for Java (Distefano and Parkinson 2008; Haack and Hurlin 2008). Unlike VeriFast, all of these tools attempt to infer loop invariants automatically. Abstract predicates were introduced by Parkinson and Bierman (2005). The details of applying separation logic to the C language were studied by Tuch et al. (2007).

Alternative specification and verification approaches, based on generation of verification conditions instead of symbolic execution, include Zee et al. (2008) and the approaches based on dynamic frames (Kassios 2006), in particular the work on automation of dynamic frames (Smans et al. 2008b,c,a), the work on regional logic (Banerjee et al. 2008), and the work on Dafny (Leino 2008).

Acknowledgments

The authors would like to thank Jan Smans for the many helpful discussions, and for co-authoring the composite.c example.

References

Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP 2008*, 2008.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In K. Lodaya and M. Mahajan, editors, *FSTTCS 2004*, number 3328 in LNCS, pages 97–109, 2004.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *APLAS 2005*, number 3780 in LNCS, pages 52–68, 2005.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2006.

James Brotheston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *POPL 2008*, 2008.

Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS 2007*, 2007.

Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. In *POPL 2007*, 2007.

Dino Distefano and Matthew Parkinson. jStar: Towards practical verification for Java. In *OOPSLA 2008*, 2008.

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS 2007*, 2007.

Christian Haack and Clément Hurlin. Separation logic contracts for a Java-like language with fork/join. In *AMAST 2008*, 2008.

Ioannis T. Kassios. Dynamic frames: support for framing, dependencies and sharing without restrictions. In *FM 2006*, 2006.

K. Rustan M. Leino. Specification and verification of object-oriented software: Marktoberdorf international summer school 2008 pre-lecture notes, 2008.

Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL 2005*, 2005.

J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS 2002*, 2002.

Jan Smans, Bart Jacobs, and Frank Piessens. An automatic verifier for Java-like programs based on dynamic frames. In *FASE 2008*, 2008a.

Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *Workshop on Formal Techniques for Java-like Programs*, 2008b.

Jan Smans, Bart Jacobs, and Frank Piessens. VeriCool: an automatic verifier for a concurrent object-oriented language. In *FMOODS 2008*, 2008c.

Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *POPL 2007*, 2007.

Hongseok Yang, Oukseh Lee, Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *CAV 2008*, 2008.

Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI 2008*, 2008.