Verification of Pointers

Nils Klarlund*& Michael I. Schwartzbach†
{klarlund,mis}@daimi.aau.dk
Aarhus University, Department of Computer Science,
Ny Munkegade, DK-8000 Aarhus, Denmark

Abstract

Our recent work links type checking in programming languages to verification based on automata. In this survey, we give an overview of our methods and suggest directions for furture rearch.

In our approach, we view data types as invariants and devise a logical and decidable framework for expressing global properties of a store consisting of records and pointers. We can express common properties, for example about doubly-linked lists and their algorithms. Such properties seemed to have called for a full Hoare logic beyond the reach of type checking and decidability.

Our work is based on monadic second-order logic. Thus verification boils down to calculations on finite-state automata. This raises specific questions about combinatorial techniques for representing state spaces if these calculations are ever to be carried out on more than simple examples.

Topics: program verification, data types.

*The author is supported by a fellowship irom the Danish Research Council.
†The author is partially supported by the BRICS Center under the Danish Research Foundation.

1 Introduction

In programming languages, data types impose invariants on the store. The store is a graph whose nodes are record values and whose edges are pointer values. Unfortunately, existing type systems offer precious little help in verifying and analyzing shapes of pointer structures. This is unfortunate since many errors are a result of inappropriate pointer manipulation. For example, it is to be expected that on a university examination in data structures, very few students would be able to correctly state an algorithm for reversing a doubly-linked list.

Since so many errors appear trivial, it seems unfair that no tool exists to assist in the construction of pointer algorithms. Even for simple text-book exercises about data structures, the state of the art in verification and type checking offers virtually no help in practice. Are these problems unassailable?

Not completely, since compilers do already offer some help in checking code. But to arrive at better analyses, we should not be intimidated by undcidability results. Our goals are, after all, more modest: provide as much automated assistance as computationally possible.

To approach this goal, we must first identify ways of formulating properties amenable to automated analyses. We have chosen monadic second order logic on trees as the vehicle for expressing interesting properties of pointers and data structures. This logic is decidable, and thus our efforts have concentrated on two issues: one is to show that pointers can be expressed and the other is to show that the changes to the store affected by a program can also be expressed.

In this survey we give an overview of our methods. We study examples that are typical for the difficulties inherent in pointer manipulations. We also discuss some of the combinatorial problems inherent in our approach and indicate directions for future research.

1.1 Traditional Approaches

In most existing languages, such as those in the **PASCAL** family, a program defines a finit collection of record types, and each pointer variable is restricted to indicate values of a single of these types. Thus, **PASCAL** types do not

capture properties of shape.

In contrast, *recursive data type* impose severe global constraints on the store, since records and pointers are restricted to form regular families of trees.

Thus, in the liberal **PASCAL** tradition, intricate structures can be programmed, but only weak invariants can be verified by the type checker. With recursive data types, stronger invariants are verified, but only simple graphs can be represented in the store.

1.2 Graph Types

In [10], we suggested *graph types* for lessening this conflict. As with recursive data types, graph types allow the store to consist of certain regular families of graphs, but more than merely trees. Our technique divides the pointers into two disjoint classes: *ordinary* and *auxiliary*. The ordinary pointers form a canonical spanning tree of every value, called the *backbone*, and are specified as values of a recursive data type. The auxiliary pointers are specified through *routing expressions*, which are regular expressions over an alphabet of *directives*. This dichotomy corresponds well to practice: ordinary pointers are the building material for lists and trees, whereas auxiliary pointers correspond to specific short cuts (such as a pointer to the previous element in a doubly-linked list) or loosely restricted pointers across the backbone (such as indices into other data structures).

Graph types extend the notion of recursive data types by allowing short cuts to be specified in type specifications. Graph types suffer two major limitations: only the backbone can be directly manipulated in programs, and auxiliary pointers always depend functionally on this backbone. Thus only short cuts can be expressed.

1.3 Invariants as Data Types

In type checking, undecidability is an important limiting factor. In [9], we exhibit a powerful, yet still decidable, framework for typing data structures. We here use invariants as data types: *data types are assertions about the entire store.* Of course, even recursive data types may be construed as sim-

ple invariants about the store. As we show in the present overview, many useful and interesting properties can be expressed and in principle be verified automatically.

1.4 The Theory

In [11], we devise a logical graph formalism based on the notion of edge constraints to specify a large class of graph families. Many graphs occurring in practice, such as trees where each node contains an extra unconstrained edge, cannot be described by any of the known context-free graph grammars, but can be described as a logical specification based on edge constraints. Also we define graph transformations, called *transductions*, which model local changes to the store.

The main result of [11] is that the problem of *transductional correctness* is decidable: there is an algorithm that given a transduction and graph specifications $\mathcal{A}$ and $\mathcal{B}$ determines whether for any graph satisfying $\mathcal{A}$, any new graph resulting from the transduction satisfies $\mathcal{B}$.

1.5 The Semantical Results

In [9], we show through some quite involved technical results that for a simple programming language, an advanced type system for pointers can be built. The programming language contains all essential pointer manipulations: allocation, deallocation, dereferencing, and assignment. In addition, the language contains limited iteration. We demonstrate that a powerful logic, the *Monadic Second-Order Logic of Recursive Data Types (M2L-RDT)*, can be used to express *assertions* about pointers. The backbone is directly described and auxiliary pointers are described as pointer constraints, which are M2L-RDT formulas of a special form. Our assertions are used to define the global picture of backbones, auxiliary pointers, pointer variables, and their mutual relations. We extend the usual syntax for type and variable declarations to include assertions about the global store and auxiliary pointers.

From this we may extract a single *well-formedness* assertion expressing the total declarative information. Type checking now amounts to preservation of validity of this assertion.

Even though the assertional language itself may seem undecidable, the main result in [9] is that the programming constructs in our language are expressible as transductions. Thus the validity of *Hoare triples* – and hence type checking – is decidable.

1.6 In This Survey

In this survey we give new examples to show that our approach is a practical and concise way of specifying some common data structures and their algorithms. The technical results in [11] and [9] are quite involved, but through this survey the reader should be convinced of what is the essence of our methods: that many substantial properties of data structures are regular in the technical sense (and in the informal sense) and therefore amenable to automated analysis.

Thus our work is similar, in spirit and technically, to the field of verification of concurrent algorithms, where a key technique is also to identify as much regularity as possible.

We envisage a system where the programmer may use the strong automated responses to gradually modify or annotate the code. We do not know to which extent such a system is feasible in practice.

But in principle, our results ensure that aspects of many different kinds of program analysis are captured for our extended straight-line code. For example, the store is completely analyzed for tag elimination, aliases, dangling references, and unclaimed memory. Whenever a triple is invalid, intelligent diagnostics can be produced in the form of a smallest counter-example consisting of a pre- and a post-store. This information may allow a programmer to correct the code or to strengthen program assertions.

For a full language involving **while**-loops our method is of course only approximate, but we do give an example in this survey showing that it may still be useful.

2 Related Work

There have been surprisingly few attempts to restrict the assertions or the programs sufficiently to obtain complete or decidable Hoare logics. One example that we know of is [4], which considers first-order assertions for a **while**-language on integers, but whose assertions and programming constructs can only express properties in Presburger arithmetic (for which decidability follows from that of second-order monadic tree logic).

The concepts of backbone auxiliary pointers are implicit in [3], which suggests a programming language construct, called a path, describing the value of an auxiliary pointer. For example, to maintain a correct description of a pointer to the last element of a list x, a variable x.path containing the selectors to be followed, i.e. the route, is updated every time the list is changed. The authors show that this use of shadow variables provides a natural reasoning style in Hoare logic.

Hyperedge-replacement grammars [5] define classes of graphs whose monadic second-order logic is decidable, but they do not capture many common data structures describable by our techniques such as trees with unconstrained auxiliary pointers leading from every cell.

Analysis of straight-line code starting in a single store or in all stores is a well-known technique from compiler optimization [1]. However, it is uncommon to allow general assertions on the pre-store and to offer a uniform framework for many kinds of analysis.

It is often possible to view type checking as program verification of simple assertions. Traditionally, other phrasings are chosen: prose, as seen in numerous manuals; deduction, as derived from logic [13, 2]; and constraints, which emphasize algorithmic aspects [12]. The verification contents become more evident as assertions become richer.

3 Specification of Stores

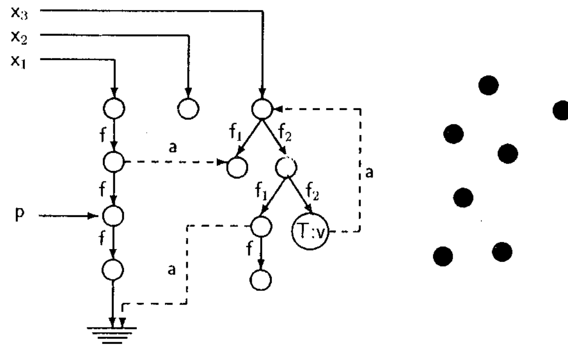
We use a simple model of records and pointers. A *store* is a directed graph whoes nodes are called *cells* and are divided into three disjoint sets:

- *record cells*, which are labeled with *types* and *variants* and have outgo-

ing edges, called *pointers*, labeled with *field names*;

- *free cells*, which correspond to the unused part of the store and have neither incoming nor outgoing pointers; and
- *a single* **null** *cell*, which has no outgoing pointers.

The pointers from record cells must lead to either **null** or other record cells, and each pointer is either *ordinary* or *auxiliary.* Usually, the graph induced by the ordinary pointers is required to form a spanning forest corresponding to the underlying recursive data types. The store is accessed through named *data* and *pointer* variables, both of which indicate cells. The only difference is that data variables always contain the roots of the spanning forest, whereas pointer variables may indicate arbitrary cells. The following is a sketch of a store:



In this example, there are three data variables x_i and a single pointer variable p. Correspondingly, there are three trees in the spanning forest. The record values are pictured as white circles and are labeled with a type T and variant v (as shown in one case), the free cells as black circles, and the single **null** cell as a ground symbol. The pointers of ordinary fields are indicated as solid arrows and auxiliary fields as dashed arrows. The backbone consists of the entire store except for auxiliary pointers. The free cells are used to explicitly model allocation and deallocation in the store. Since we only consider shapes, the model abstracts away from values such as integers and booleans.

3.1 Our Approach

We use monadic second-order logic on trees to specify the stores. The challenge is of course to discover a syntactic formalism that permits the specifications to be as intuitive and familiar as ordinary type specifications.

The underlying backbones are specified through ordinary recursive data types. We view such a type as abbreviating an involved predicate on stores imposing restrictions on their connectivity and labeling. This predicate can always be written out in monadic second-order logic.

A major theoretical obstacle is that auxiliary pointers cannot be mentioned directly in the logic. If this was allowed, then undecidability would follow. To circumvent this problem, we have devised a more indirect technique with predicates involving two extra variables **src** and **dst** indicating the source (i.e. the node containing the pointer field) and destination of an auxiliary pointer (i.e. the node that is pointed to).

We have developed many helpful notations for expressing such predicates. The regular *routing erpressions* from [10] are convenient; for example, post-order traversal of a tree is easily expressed. The "ultimate" syntax will only be shaped through extended practical experience, which we have yet to fully gain. Note that the meaning of such specifications — regardless of syntactic sophistications — always boils down to regularity: a single well- formedness predicate on stores expressed in our extension of monadic second-order logic on trees.

Example: List with Designated Element

The type H of linear lists in which the header contains an anxiliary pointer to *some* element of the list is sketched as follows:

type H → (first: L, some: "*leads to some node below* first")
type L → (fhead: Int, next: L)
→ **null**

The backbone is generated by the underlying recursive data type denoted by the first, head, and next fields; hence, it is a list. Note that the type L has two variants, one of which is **null**. To specify the predicate on the auxiliary

pointer some, we use the routing expression **src**⟨first.next*⟩**dst**, which states that **src** and **dst** must be related by the routing expression first.next*. This means that the destination can be reached from the source by following a path of the form first.next...next. Thus, the formal specification is as follows:

type H → (first: L, some: **scr** ⟨first.next*⟩ **dst**)
type L → (head: Int, next: L)
→ **null**

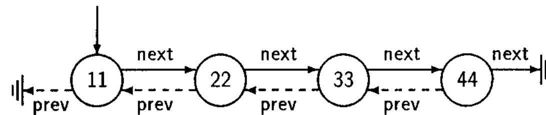
Simple routing expressions capture many interesting data types, such as cyclic lists, leaf-to-root-linked trees, leaf-linked trees, and threaded trees. Our work on graph types [10] focused on the special cases where auxiliary pointers are functionally determined by the backbone, such as in the next example.

Example: Doubly-Linked Lists

Let us return to doubly-linked lists, which we specify as follows:

type D → (head: Int, next: D, prev: (^**scr** ∧ **null**? **dst**)∨(**dst**.next=**src**)))
→ **null**

Here the pointers of the next fields span the backbone. The prev field is auxiliary and the pointer must satisfy: the source is the first node (^**scr**) and the destination is **null** (**null**?**dst**) or the next field of the destination points to the source (**null**.next=**src**). A typical value is:



Example: Trees with Blue and Green Leaves

As a final and more involved example, consider the recursive data type C consisting of binary trees whose leaves are either blue or green. We wish to

have a data variable x of this type and a pointer variable p that always points to a blue leaf of x or to the root if no such leaf exists. The pointer variable p is declared with a formula that constrains its possible destinations.

```
type C → (left, right: C)
       → blue()
       → green()
var x:  C
var p:  x⟨↓*.blue?⟩dst ∨ (¬(∃α : x⟨↓*.blue?⟩α) ∧ x?dst)
```

The type C has three variants two of which are named blue and green. There are two disjoint cases for p. The first ($x\langle\downarrow^*.blue?\rangle\mathbf{dst}$) is to start at the root of the data variable x and follow a downwards path ($\downarrow^*$) until a node of variant blue is reached. The other is a conjunction of two parts. The first part ($\neg(\exists\alpha : x\langle\downarrow^*.blue?\rangle\alpha)$) is the negation of an existential formula (quantifying over cells) and expresses that x does not have any blue leaves. The second part ($x?\mathbf{dst}$) states that the destination must be the root of x.

4 Verification of Programs

We now have a technique for specifying interesting data types as predicates on stores. The next step is to provide a programming language for manipulating such stores. We have pursued two approaches:

1. In the case of graph types, it sufficies to use ordinary operations on recursive data values, since the auxiliary pointers are functionally determined by the backbone and can be automatically updated at run-time. See [10] for more details.

2. A more general approach is to use an ordinary imperative **while**-language with the usual pointer manipulations. The challenge is to type-check such programs.

In [9] we show how a substantial part of this programming language can be translated into the graph transductions introduced in [11]. Our decidability result for transductional invariance [11] in principle allows completely

automatic verification of Hoare triples of programs written in this restricted language. For obvious reasons, the fragment we can handle is not Turing complete; it is basically straight-line code extended with certain regular control strutures, which are also described as routing expressions For full **while**-loops, we must resort to well-known verification techniques; however, once a loop invariant has been phrased, then the decidability of Hoare triples accomplishes the remaining task automatically.

Example: Construction of List with Designated Element

To construct a list y with four elements and the second being designated, we use the code:

```
type H → (first: L, some: src⟨first.next*⟩ dst)
type L → (head: Int, next: L)
       → null
var x : L
var y : H

x := L(11,L(22,L(33;L(null))))
y := H(x,x → next.next)
```

These constructors require no further assertions in order to be automatically verified.

Example: Reversal of Doubly-Linked Lists

Even when auxiliary pointers are functionally dependent on the backbone, the code that one can write based solely on the operations of recursive data types is often too inefficient. For example, if we want to reverse doubly-linked lists, then the easy recursive traversal algorithm involves copying of record cells. Instead, we would like to reverse pointers in place.

We indicate next how our decidable Hoare logic is used to verify the rather messy details of such an algorithm. Consider again the type of doubly-linked lists:

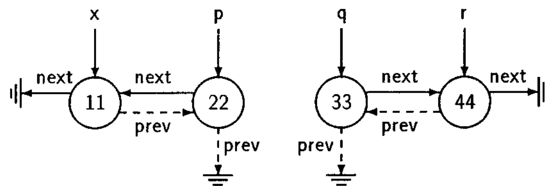
Type D $\rightarrow$ head: Int,next: D, prev: ($\hat{}$**src** $\wedge$ **null**?**dst**) $\vee$ (**dst**.next = **src**))
null

We wish to reverse the value of a variable x of type D. A complete verification of such a program involves arithmetical properties that goes beyond our logic. e.g. the fact that the length of the list remains the same. However, we can automatically verify that we maintain well-formedness of shape. This is a necessary requirement for correctness and also a finely masked filter for many errors. The proposed program looks as follows.

```
var x: D
var p,q,r: D? dst

p :=null;
q := x;
while{D-wf? p ∧ D-wf? q}
      ¬null? q do
              ¬null? q do
              r := q → next;
              q.next := p;
              if¬null? p then p.prev := q end;
              if¬null? r then r.prev := null end;
              q.prev := null;
              p := q;
              q := r
end
x := p
```

Here p, q, and r are pointer variables that point to records of type D (D?**dst**). Intuitively, the algorithm works according to the picture:



where p indicates the part of the list that has already been reversed and q indicates the remaining part. The formal invariant ({D-wf?p$\wedge$ D-wf?q})

states only that p and q point to well-formed D-values. Our algorithm can verify this code, and in particular we may conclude that x is a well-formed value of type D upon completion of the loop.

Observe that if any of the assignment statements or their mutual order is corrupted in the trivial manner that so often occurs, then the code could no longer be verified and counter-examples could be provided. Note that *during* an iteration of the loop, the invariant does not hold. However, the verification algorithm collects sufficient information to determine that the invariant is restored after each complete iteration. In particular, the store suffers from neither unclaimed garbage nor dangling references.

Note that we have included the statements **if** ¬**null**? r **then** r.prev := **null** **end** and q.prev := **null** to maintain the invariant. The first of these could be omitted and the second could be replaced by **if** ¬**null**? p **then** p.prev := **null** **end** after the loop. In that case, we would modify the invariant so that it states: with the modification of the store corresponding to the two statements, well-formedness holds.

4.1 Example: Updating Trees with Blue and Green Leaves

This last example shows how concrete counter-examples may be generated. Recall the blue-green trees:

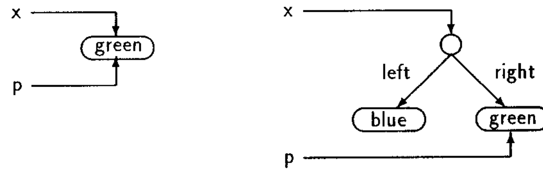
```
type C → (left, right: C)
       → blue()
       → green()
var x:  C
var p:  x⟨↓*.blue?⟩dst∨(¬(∃α : x⟨↓*.blue?⟩α)∧ x?dst
```

At some point, the transformation:

```
x := C(left: blue()), right: x)
```

which extends x with a blue left-sibling, might seem reasonable. However, we have committed an archetypical and notoriously subtle error. The above code

is rejected by the type checker, which offers the following pre- and post-store demonstrating what could go wrong: See next figure.



We see that the problem is the pointer p, which must point to a blue leaf or to the root. This holds in the pre-store, but in the post-store the indicated node is neither blue nor is it the root.

5 Where Does Decidability Come From?

Already around 1960, it was discovered that usual regular languages have a logical characterization (for references, see [14]) in terms of a monadic second-order logic. The fundamental correspondence is that to any formula, open or closed, there is an automaton that recognizes the set of all interpretations (suitable encoded) that satisfy the formula. The calculation of these automata involves cross product construction (for $\wedge$ and $\vee$), determinization (for $\neg$), and projection (for $\exists$). In addition, the Myhill-Nerode theorem is used in practice to always keep the automata as small as possible.

This correspondence also holds for the monadic second-order logic on trees, which allows second-order quantification over sets of nodes.

The decision procedures have recently been implemented in prototype tools at our department and at the University of Kiel. The theory, the decision procedures are nonelementary, since each quantifier alternation introduces an exponential state space blowup. In practice, quantifier alternation is bounded, but the decision procedures still have an hyper-exponential lower bound, if for example, there are two quantifier alternations (such as $\forall\exists\forall$).

Our techniques

The monadic second-order logic of recursive data types is undecidable when interpreted over arbitrary stores (not necessarily tree-formed) since even the

first-order theory of finite graph is undecidable. However, when restricted to trees, the logic becomes decidable. After a transformation by a program, the store is no longer tree-formed. But for the programs we consider, the changes can be described inside the logic itself. In fact, our results in [11] show that even the auxiliary pointers can be treated in a similar way, although they cannot be directly described in the logic.

6 Directions for Future Work

Our formalism shows that regularity—in the technical sense—is inherently present in many data structures and their algorithms.

From our point of view, we see two important challenges in verification theory:

- Look for more methods of identifying regularity.

  Identification of regularity can be seen a state space reduction (from usually infinite spaces) to the finite state spaces of automata. It might be possible to weaken sufficiently the semantics of Hoare logic such that errors about data structures with infinite ranging values can also be formulated in a decidable framework. As with the work discussed here, the main observation is that assignment statements only affect the store locally. Thus reasoning about "local errors" should be possible to some extent, since undecidability stems from considering tilings or the like of unbounded global spaces. One example of a local error is if two elements in an ordered list are not in the right order.

  Two other recent examples of identifying regularity that we have been involved with are type inference [12], which build on complex reasoning about infinite systems that turn out to be regular, and verification of simple parameterized concurrent systems [7], where invariants on an unbounded number of processors are verified automatically by a system based on second-order monadic logic.

- Look for more methods of reducing finite state spaces.

If the representations of automata remain small, the decision procedures that we suggest work in practice, sec [7]. Usually, the initial description is small and a possible counter-example would also be small.

Thus a major open question is whether we can avoid explicitly constructing big product spaces during the calculations. For example, a frequent problem in finite-state verification is the state explosion that results from logical structures of the form $\exists x_1, \ldots, x_n : P_1 \wedge \cdots \wedge P_m$, where the P_js are represented by non-deterministic automata. (A disjunction inside the existential quantifier would be handled by distributing, the quantifier over the disjunction). If the formulas share no variables, then it is exponentially more efficient to first determinize each P_j and then form the product than to do it the other way round. But what if information is shared among the P_j through common variables? The only general technique is to form the non-deterministic product space and then determinize a doubly exponential endeavor.

A verification method based on explicit formation of product spaces cannot deal with this situation in practice. A similar problem has been addressed in [6] . Possible venues solving such problems may also be based on the implicit product spaces of asynchronons automata in trace theory; see [8] for a determinization construction. Unfortunately little is known about reducing the state spaces of such automata.

References

[1] A. V. Aho and .J. D. Ullman.
Principles of Compiler Design.
Addison-Wesley, 1977.

[2] L. Cardelli.
Typeful programming. Technical Report No. 45,
Digital Equipment Corporation, Systems Research Center, 1989.

[3] R. Cartwright, R. Hood, and P. Matthews.
Paths: An abstract alternative to pointers.

In *Proc. 8th ACM Symp. on Princ. of Programming Languages*, pages 14-27, 1981.

[4] J. C. Cherniavsky and S. N. Kamin.
A complete and consistent Hoare axiomatics for a simple programming language.
JACM, 26:119-128, 1979.

[5] B. Courcelle.
Graph rewriting: An algebraic and logic approach.
In *Handbook of Theoretical Computer Science, Elsevier*, pages 193-242, 1990.

[6] A.J Hu and D.L. Dill.
Efficient verification with BDDs using implicitly conjoined invariants.
In *Computer Aided Verification 1993, LNCS 697*, 1993.

[7] M.E. Joergensen, J.L. Jensen, and N. Klarlund.
Practical uses of monadic second-order logics on strings.
In preparation, 1994.

[8] N. Klarlund, M. Mukund, and M. Sohoni.
Determinizing asynchronous automata.
Submitted, 1994.

[9] N. Klarlund and M. Schwartzbach.
Data types as invariants.
unpublished, 1993.

[10] N. Klarlund and M. Schwartzbach.
Graph types.
In *Proc. 20.th Symp. on Princ. of Prog.Lang.*, pages 196-205. ACM, 1993.

[11] N. Klarlund and M. Schwartzbach.
Graphs and decidable transductions based on edge constraints.
In Proc. CAAP'94 (TAPSOFT), 1994
To appear.

[12] D. Kozen, J. Palsberg, and M. I. Schwartzbach.
Efficient inference of partial types.

Journal of Computer and System Science.
To appear. Also in Proc. FOCS'92, 33rd IEEE Symposium on Foundations of Computer Science, pages 363-371,
Pittsburgh, Pennsylvania, October 1992.

[13] R. Milner.
A theory of type polymorphism in programming.
Journal of Computer and System Sciences, 17:348-375, 1978.

[14] W. Thomas.
Automata on infinite objects.
In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. pages 133-191. MIT Press/Elsevier, 1990.