Lazy Contract Checking for Immutable Data Structures

Robert Bruce Findler, Shu-yu Guo, and Anne Rogers

University of Chicago
{robby,arc,amr}@cs.uchicago.edu

Abstract. Existing contract checkers for data structures force programmers to choose between poor alternatives. Contracts are either built into the functions that construct the data structure, meaning that each object can only be used with a single contract and that a data structure with an invariant cannot be viewed as a subtype of the data structure without the invariant (thus inhibiting abstraction) or contracts are checked eagerly when an operation on the data structure is invoked, meaning that many redundant checks are performed, potentially even changing the program's asymptotic complexity.

We explore the idea of adding a small, controlled amount of laziness to contract checkers so that the contracts on a data structure are only checked as the program inspects the data structure. Unlike contracts on the constructors, our lazy contracts allow subtyping and thus preserve the potential for abstraction. Unlike eagerly-checked contracts, our contracts do not affect the asymptotic behavior of the program.

This paper presents our implementation of these ideas, an optimization in our implementation, performance measurements, and a discussion of an extension to our implementation that admits more expressive contracts by loosening the strict asymptotic guarantees and only preserving the amortized asymptotic complexity.

1 Introduction

Assertion-based contracts play an important role in constructing robust software. They give programmers a technique to express program invariants in a familiar notation with familiar semantics. Contracts are expressed as program expressions of type boolean. When the expression's value is true, the contract holds and the program continues. When the expression's value is false, the contract fails, causing the contract checker to abort the program, identify the violation, and blame the violator. Identifying the faulty part of the system helps programmers narrow down the cause of the violation and, in a component-oriented setting, exposes culpable component producers.

Contracts enjoy widespread popularity. For example, contracts are currently the second most requested addition to Java.[1] In C code, assert statements are particularly popular, even though they do not have enough information to assign blame properly and thus are a degenerate form of contracts. In fact, 60% of the C and C++ entries to the 2005 ICFP programming contest [9] used assertions, even though the software was produced for only a single run.

[1] http://bugs.sun.com/bugdatabase/top25_rfes.do, as of Groundhog Day, 2007.

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 111–128, 2008.

Despite the popularity of contracts, the state of the art in contract checking for data structures is poor. In order to use contracts on data structures, programmers are forced to choose between copied code (and thus doubled maintenance costs) and very poor performance (often infeasible, as we show). Our contract checker provides a new alternative. It is designed to strike a balance between performance and the amount of checking, motivated by the desire to avoid changing the asymptotic complexity of operations that have contracts. Our implementation is written in PLT Scheme [13], and is applicable to other strict languages with immutable data structures.

The next section uses binary search trees to make the programmer's existing poor choices plain. Section 3 explains the design of our contract checker and how it limits the amount of checking performed, in order to recoup tractable performance. Since our design is partially motivated by performance, we spend Section 4 explaining our implementation and Section 5 presenting some performance measurements that validate our design. For example, an experiment on binomial heaps show that eager checking may cause the program to be 2,000 to 20,000 times slower, while our lazy contract checker reduces that overhead to a factor between 8 and 10. Section 6 discusses an extension to our contract checker that relaxes the strict asymptotic complexity requirements; by giving the contract checker the freedom to preserve only the amortized complexity, we gain the ability to write more expressive contracts. Section 7 discusses related work and Section 8 concludes.

2 A Rock and a Hard Place

To see how existing techniques for data structure contracts fail programmers, consider a binary search tree library (shown in Figure 1) that is built on top of a binary tree library.

```
(module bt mzscheme
  (define-struct node (n left right))
   ...
  (provide (struct node (n left right)) marshal-bt unmarshal-bt))

(module bst mzscheme
  (require bt)
  ;; a Binary Search Tree (bst) is either null or
  ;; (make-node number[n] bst[left] bst[right])
  ;; where the numbers in left are less than (or equal to) n
  ;; and the numbers in right are greater (or equal to) n

  (provide find-bst) ;; : bst number → boolean
  (define (find-bst t n)
    (and (node? t)
         (or (= n (node-n t))
             (and (< n (node-n t))
                  (find-bst (node-left t) n))
             (and (> n (node-n t))
                  (find-bst (node-right t) n))))))
```

Fig. 1. Binary search trees, without contracts

The binary tree library is left mostly to the reader's imagination, but a skeleton is shown in the `bt` module.[2] It exports basic operations on binary trees (marshaling them to and from disk) and a node record for building and querying the nodes in a binary tree. In PLT Scheme, records are called structs. The **`define-struct`** introduces a new struct that consists of three fields. It also defines five functions: `make-node` used to build new nodes, `node?` used to recognize node structs, and `node-n`, `node-left`, and `node-right` used to extract the fields from a node struct. In general, a struct definition introduces a single maker, a single predicate, and one selector per field. The **`provide`** clause exports the struct and the marshaling functions.

The `bst` module requires the `bt` module and defines a binary search tree data structure in a comment, according to the discipline of *How to Design Programs* [7]. The comment specifies that binary search trees have the same shape and use the same node struct as binary trees, but also have the binary search tree invariant. The programmer carefully uses the same basic data structure so that the existing library for binary trees (marshaling and unmarshaling functions in this case) can also be used with binary search trees. Beyond the data definition, the `bst` module also provides `find-bst`, a function for finding numbers in binary search trees that takes advantage of the binary search tree invariant to avoid the recursive calls when it is safe to do so.

As the program grows from a little script to a part of a robust application, its author decides to improve the reliability of the program by writing a checkable contract on the data structure as shown in Figure 2. The `bst?` predicate uses the `bst-between?` helper function to test whether its input is a binary search tree. The function `bst-between?` enforces the binary search tree invariant using two accumulators, a lower and upper bound on the values in the tree. The accumulators are initially negative and positive infinity respectively, and as the traversal passes each interior node, the bounds tighten in the recursive calls.

Finally, the `bst?` predicate is used in the contracts for the provided functions.[3] The contract on `find-bst` is an $\rightarrow$ contract and is written using prefix notation. The last argument to $\rightarrow$ is a predicate on the result of `find-bst`, ensuring that it always produces booleans. The other two arguments are predicates on the inputs to `find-bst`, ensuring that the first argument is a binary search tree and that the second argument is a number. Similarly, the contract on `bst?` ensures that it is a predicate function.

Although it may not be obvious at first glance, the binary search tree portion of the revised library is now completely useless. In particular checking `find-bst`'s contract means that the `bst?` predicate is called on each argument supplied to `find-bst` in order to enforce the pre-condition (domain) contract. Since `bst?` traverses the entire tree, it ruins the optimization built into the `find-bst` function, changing the asymptotic complexity from logarithmic to linear, an exponential slowdown.

This state of affairs leaves the programmer in a bind; both the loss of performance and the loss of reliability are unacceptable. The conventional solution to this problem is to hide the raw struct operations behind an opaque module boundary and only export

[2] The `mzscheme` that appears after the module name is the language name of the module. MzScheme is PLT Scheme's implementation of the Scheme language.

[3] We use the PLT Scheme contract library's notation [22] throughout. Support for lazy data structure contracts was added to PLT Scheme's contract library in v350 (released June 2006).

```
(module bst mzscheme
  (require (lib "contract.ss"))

  ;; bst? : any → boolean
  (define (bst? t) (bst-between? t −∞ +∞))
  (define (bst-between? t low high)
    (or (null? t)
        (and (node? t)
             (number? (node-n t))
             (≤ low (node-n t) high)
             (bst-between? (node-left t) low (node-n t))
             (bst-between? (node-right t) (node-n t) high))))
  (define (find-bst t n) ...) ;; as in Figure 1

  (provide/contract
   [find-bst (→ bst? number? boolean?)]
   [bst? (→ any/c boolean?)]))
```

Fig. 2. Binary search trees, with contracts

operations that guarantee the binary search tree invariants (*e.g.*, self-balancing insert plus an empty binary search tree). Of course, this non-solution has the problem that a client of `bst` module cannot reuse the `bt` operations on `bst`s.

A programmer may attempt to work around this by providing new versions of each of the `bt` operations that simply unwrap a `bst` struct, apply the operation, and then rewrap it. This approach is not desirable for two reasons. Not only must the `bst` programmer anticipate all future extensions to the `bt` library, he must now also verify that none of the `bt` operations violate the binary search tree invariant, rather than letting the system itself ensure the binary search tree invariant holds.

Another solution is to provide injection and projection functions that convert binary search trees to and from binary trees and, along the way, verify the invariant. This solution amounts to changing the pre-condition on the `find` operation to a simple check, but requiring that programmers rewrite their programs to decide explicitly where to do the real checks. Worse, it is not always possible to avoid an asymptotic slowdown when binary search tree operations are interleaved with binary tree operations.

In general, code reuse is enabled by the ability to view a data structure with an invariant (like the binary search tree) as the same data structure but without the invariant. Or, put another way, code reuse is hindered by taking that ability away or allowing it only when accompanied by expensive invariant checks. Thus, the goal of this work is to provide a form of contract checking that allows programmers to view data structures with invariants as if they are just the underlying data structures, without any special action on the part of the programmer and without violating the invariant.

Throughout the remainder of this paper, we continue to use binary search trees as a motivating example. Nevertheless, our technique applies to many data structures that have invariants: heaps, self-balancing trees, sorted lists, etc. It is also useful whenever one wishes to use refinement types (but when a refinement type checker is not strong enough) such as even-length or non-empty lists, or viewing the result of Scheme's

`read` as having a particular shape. Another particularly fertile ground is a compiler's intermediate representation. Well-known intermediate representations like CPS and A-normal form [12] are easily expressed as contracts over the general expression type, and compiler authors who take advantage of them can determine which pass of a compiler has failed when bad output is produced.

3 Lazy Contract Checking

Our solution to the problem presented above is to introduce a new kind of contract for data structures to be used with the existing contract combinators in PLT Scheme. These contracts have the benefit of the contracts in Figure 2, namely they permit the programmer to use a single value with multiple, different contracts, but instead of eagerly checking the entire data structure when checking a contract, our contracts lazily check the portions of the data structure that the function inspects, as it inspects them.

Our contracts extend PLT Scheme's **`define-struct`** to **`define-contract-struct`**. It has the same syntactic shape as **`define-struct`**, but in addition to introducing a maker, predicate, and selectors, it also introduces a contract constructor. For example, the declaration

```
(define-contract-struct node (n left right))
```

introduces `node/dc`, the constructor for *node dependent contracts*. Its shape is

```
(node/dc [n contract-expr]
         [left (n) contract-expr]
         [right (n) contract-expr])
```

where each clause specifies the contract on the respective field. The `(n)` in the `left` and `right` contract specifications indicates that the contracts for the `left` and `right` fields depend on the value of the `n` field (the variables in the parenthesis are ordinary bound variables, but their names must match the names of other fields of the struct; that is, they may not be α-renamed). In general, the contract on any field may depend on any of the fields before it, but the dependencies must be specified explicitly by the programmer. Of course, `node/dc` is just one instance of a contract constructor; each **`define-contract-struct`** declaration introduces its own dependent contract constructor that expects as many fields as there are in the struct.

Using `node/dc`, the contract for a binary search tree is written as:

```
;; bounded-bst : number number -> contract
(define (bounded-bst lo hi)
  (or/c null?
        (node/dc [n (between/c lo hi)]
                 [left (n) (bounded-bst lo n)]
                 [right (n) (bounded-bst n hi)])))
(define bst (bounded-bst −∞ +∞))
```

The `or/c` contract combinator accepts any number of contracts (or simple predicates) and checks that at least one of them holds. The `between/c` contract combinator accepts

two numbers and returns a contract that matches numbers in those bounds. The contract on the left and right sub-trees of an interior node are built by recursively calling `bounded-bst` with different bounds on the values in the tree. The initial contract on a binary search tree is built by calling `bounded-bst` with negative and positive infinity.

The remainder of this section explains how dependent struct contracts behave, continuing to use the binary search trees example.

3.1 Checking During Traversal

The contract checker only checks struct contracts as the program itself inspects the data structure. To see how this plays out, consider this binary search tree and call to `find-bst`.

```
(define a-bst (make-node 5
                (make-node 3
                 (make-node 1 ...)
                 (make-node 6 null null))
                (make-node 7 ...)))
(find-bst a-bst 4)
```

The series of diagrams in Figure 3 shows the evolution of the contracts as `find-bst` traverses `a-bst` seaching for `4`. To represent the contract on the tree, we draw a box around the tree and annotate the box with the contract. So, when the tree is first passed to `find-bst`, it picks up the binary search tree contract and is labeled "$(-\infty,+\infty)$", meaning that the elements in the tree must be between $-\infty$ and $+\infty$, corresponding to the contract obtained by calling `(bounded-bst` $-\infty$ $+\infty$`)`. The first step `find-bst` takes is to examine the top node in the tree. At the point when `find-bst` first extracts a field of the top `node` struct, the contract checker steps in and verifies that the values of the fields of the node match the contract. Verifying that the number in the tree is in the appropriate range is a simple check, but to ensure that the subtrees match their contracts, the contract checker creates new boxes to avoid exploring more of the tree than the program does, as shown in Figure 3(b).

The labels on the new boxes indicate the new contracts, derived from the binary search tree invariant (as implemented by `bounded-bst`). The left sub-tree's elements

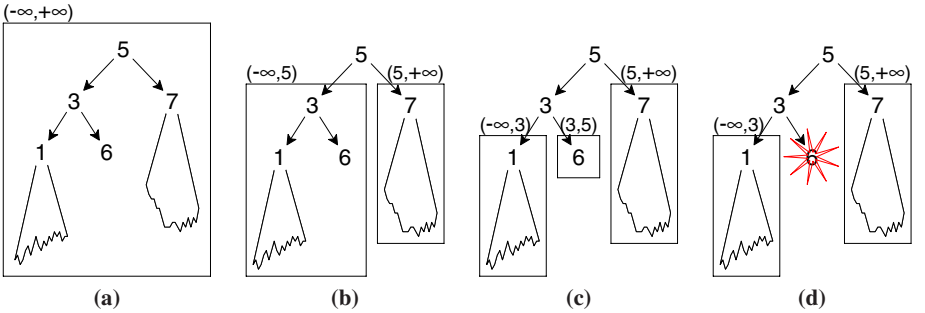


Fig. 3. Evolution of contracts during traversal of tree

must be smaller than `5` and the right sub-tree's elements must be larger than `5`. Figure 3(c) shows the state of the tree after `find-bst` inspects the left child of the root. Again, the contract checker verifies that the node's value is appropriate and creates new boxes for the sub-trees. At this point in the program, no contract violation is signaled, because the program has not yet discovered the contract violation lurking one level down in the tree. Indeed, if the program never explores that part of the tree, a contract violation will never be signaled. But, because `find-bst` is searching for a `4`, it does inspect that node, and a contract violation is signaled blaming the caller of `find-bst`.

3.2 Redundant Contracts

Although the boxes help eliminate much of the redundant work that eager contract checking would incur, it is still possible to do too much work. In particular, we must be careful to avoid accumulating multiple, redundant boxes on the same tree. To see how this happens, imagine that a tree is built up via an `insert : bst number` $\rightarrow$ `bst` operation that first calls `find-bst` to see if the value is in the tree and, if so, just returns the original tree. Consider the effect of these two calls during the evaluation of `(insert (insert a-bst 5) 5)`. Even though the two calls do not change the tree, a naive strategy for putting boxes on contracts accumulates surprisingly many new boxes.

Figure 4 shows what would happen for those two calls. Initially, the tree has no contracts, but as soon as it is passed to `insert`, the binary search tree contract is wrapped around it, as shown in Figure 4(a). The first thing `insert` does is pass the tree to `find-bst`, along with `5`. Since `5` is in the root node of the tree, `find-bst` triggers the checking of only the first layer of the contracts, pushing contracts down to the left and right sub-children, and removing the outer layer of contracts. After that, the first call to `insert` returns and its post-condition adds another box around the entire tree and we are left with the tree shown in Figure 4(b).

As the second call to `insert` happens, the pre-condition adds another wrapper to the tree, leaving us with Figure 4(c). When `insert` calls `find-bst`, it inspects the top portion of the tree, pushing both of the contracts to its subtrees, and then the post-condition of `insert` adds yet another contract outside the tree, leaving us with Figure 4(d).

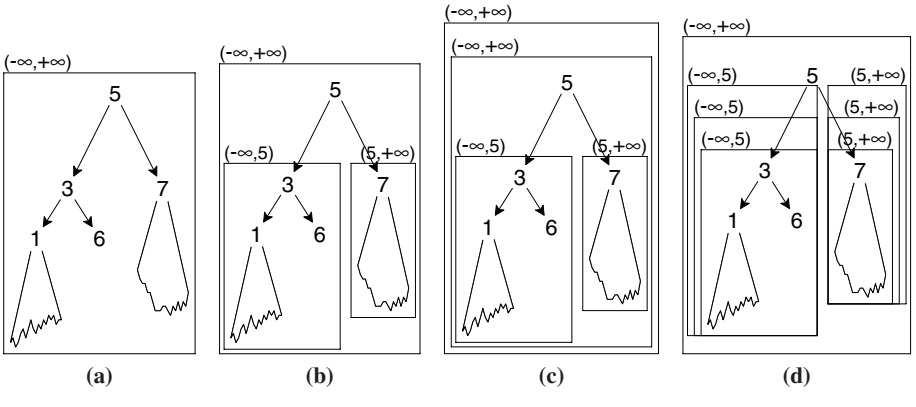


Fig. 4. Evolution of contracts during tree traversal without stronger check

To avoid this accumulation, we must be able to detect redundant contracts. In the case of a binary search tree, we can simply compare the bounds. If the box around a tree has the same (or tighter) bounds than what the new box would, then we can just leave the tree alone, relying on the existing contract to guarantee that the new contract holds.

To detect redundant contracts in general, our contract system supports a partial ordering on contracts that is used to compare two contracts to determine if one is stronger than or equal to the other. The ordering is tied to the particular contracts that our system supports. Each contract knows how to compare itself to certain other contracts in our system; if the contract does not recognize the other one, we avoid unsoundness by assuming that neither contract is stronger than the other.

As a design principle for our system, we decided that programmers who merely use contracts should not have the responsibility of specifying the stronger relationships. Instead, that responsibility should lie with the programmers that implement the contract combinators (such as `between/c`, $\rightarrow$, or the struct contracts). Accordingly, the stronger relationships are set in stone once a particular contract combinator has been defined. So far, this method has worked well enough for us, but we may also eventually investigate separating the stronger relation definition from the contract combinators and allowing programmers to extend it.

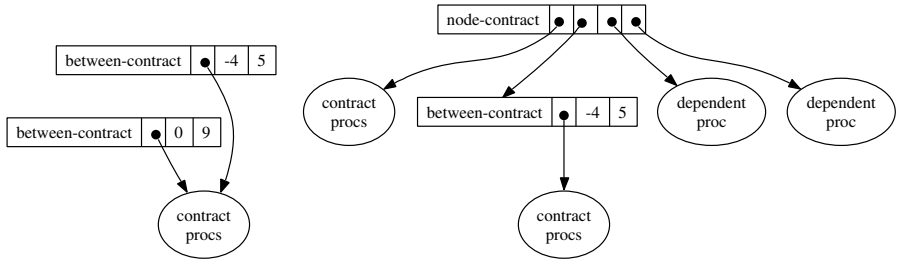
For `between/c` contracts, our system treats the one that accepts the same or a narrower range of numbers to be the stronger contract. One contract on a struct is stronger than another if the contracts on the fields of the first are stronger than the contracts on the fields of the second. Comparing function contracts uses the usual contra-variant ordering. To date, simple structural equality of contracts, combined with the bounds checking of `between/c` has been sufficient for all of the data structure invariants we have encountered (including all those in Okasaki's book [18] and in Cormen, Leiserson and Rivest [6]).

To exploit our new relation on contracts, we simply avoid adding a new contract if the contract already on the data structure is stronger than or equal to the new contract. Note that we do not need to consider blame here, unlike the case when the existent contract is not stronger; indeed, if two contracts surround a single data structure, the inner contract is always checked before the outer one, because the inner contract was placed on the object first. If the contract already on the data structure is stronger than the new contract, it does not matter who might be blamed if the new contract were to be violated; the existing contract guarantees it never fails.

Once we avoid adding redundant contracts, calling `insert` as above (or even arbitrarily many more times) would result in the wrappers shown in Figure 4(b). That is, each sub-tree would only have a single wrapper, no matter how many times `insert` is called.

4 Implementation and an Optimization

In our implementation, each contract is represented as a struct that has at least one field. That field contains a reference to a group of functions specific to that kind of contract that interpret the values in the other fields. The representation is inspired by the way



(a) Two between contracts **(b) A node contract**

Fig. 5. Contract layouts

objects are represented in class-based object-oriented languages: the record of functions is like the method table and is shared among every contract of a particular kind. As an example, Figure 5(a) shows a box-and-line diagram for the result of `(between/c −4 5)` and `(between/c 0 9)`. Each points to the same record of functions and has two numbers indicating the range it accepts.

A contract on a struct also has a shared record of contract procedures, but in addition it has one field per struct field. Each of those fields is either a contract that the contents of the field must satisfy directly, or it is a function that accepts the values in the other fields and returns such a contract. As an example, the contract

```
(node/dc [n (between/c −4 5)]
         [left (n) (bounded-bst −4 n)]
         [right (n) (bounded-bst n 5)])
```

is shown in Figure 5(b). The first field is the record of functions. Because the contract on the `n` field does not depend on other contracts, the second field of the contract record is the `between/c` contract. But, the `left` and `right` fields depend on the value of the `n` field, so they are functions that consume the `n` field's value and produce contracts.

Each contract's record of functions includes three functions. The first accepts the contract record and a value and enforces the contract. The second accepts two contracts and returns a boolean indicating whether the first is stronger than the second or not. The third function in the contract accepts a contract record and builds a name for the function to be used in error reporting.

To support lazy data structure contracts, we must not examine the struct's fields right away. Accordingly, the checking function for structs merely verifies that the struct's type matches, and then pairs the contract with the struct. Later, when a selector is applied to the struct, the contract is checked. Figure 6 contains a series of box and pointer diagrams that illustrate this process. The first diagram shows an example binary search tree, where the `null`s representing the empty tree are written `mt` to clarify the figure.

Figure 6(b) shows the tree paired with a between contract in a node-wrap struct. The node-wrap struct that holds the pair has a number of extra fields. The first field refers to the original object, but is also used as flag to indicate if the top row or the bottom row of fields are active. In the case shown, because that first field contains a reference to a struct, the top fields are active. Those fields contain a pointer to the contract, and two

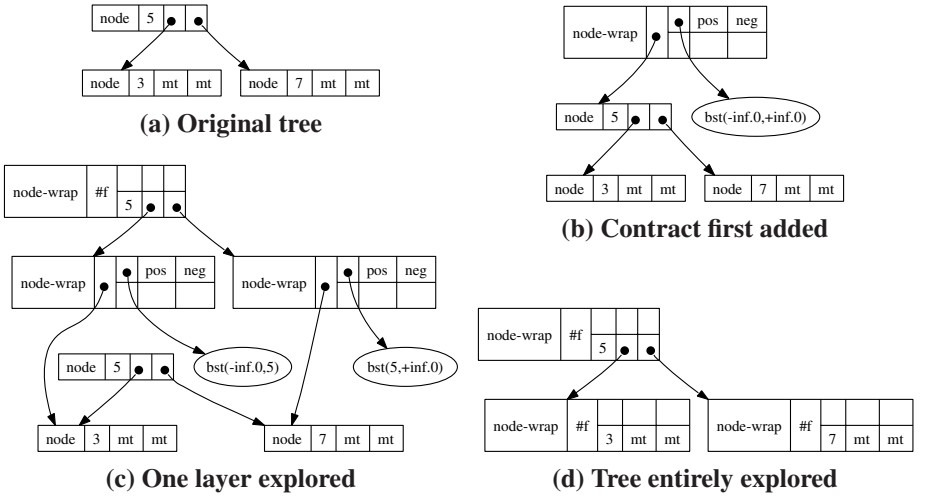


Fig. 6. Evolution of objects during contract checking

names that indicate who is to blame for contract violations. The label `pos` indicates who is to be blamed if this contract fails to hold, and `neg` is only used to support contract checking of functions that may appear inside this structure. It indicates the name of the party responsible for inputs to those functions [10,11]. Positive and negative infinity are written as `+inf.0` and `-inf.0`.

The other fields are used to implement the removal of the boxes described in Section 3. In particular, once the contract has been checked we know that it will continue to hold for all time, because the data structure is immutable. Accordingly, we place the contracted versions of the fields of the original struct into the bottom row of the node-wrap, to avoid recomputing them. When that happens, we also change the first field to `#f` in order to indicate that the bottom row is active.

Figure 6(c) shows the same tree, but after a selector has been applied to the struct with the contract, causing the contracts on the fields to be checked. The top node-wrap struct in this diagram is the same node-wrap struct in the top of diagram (b), but now the lower fields are active. The second field (in the bottom row) in that structure is `5`, the contracted version of the first field in the original struct. The final two fields are the contracted versions of the left and right sub-trees. The left sub-tree now has the contract `(bounded-bst` $-\infty$ `5)`, so it is a node-wrap struct whose first field is not `#f`. This node-wrap's top row is active, because its contract has not yet been checked. Similarly, the right sub-tree now has the unchecked `(bounded-bst 5` $+\infty$`)` contract. Finally, the fourth diagram shows the tree after all of the contracts have been checked. At this point, the tree is very similar to the original tree.

Since the top row and the bottom row are never simultaneously active for any given node-wrap struct, our implementation only has a single set of fields and uses the second field to indicate how to interpret the remaining fields.

Generally speaking, supporting the stronger relation for contracts is simply a matter of inspecting the structure of the contracts. For example, seeing if one `between/c`

contract is stronger than another amounts to comparing the numbers inside the contract record. The only exception to this is dependent struct contracts, when the fields actually are dependent. In that case, the dependent contracts are represented as functions that accept field values and return ordinary contracts. For example the `left` field of a node in the binary search tree contract is represented as the function

```
(λ (n) (bounded-bst lo n))
```

To compare such contracts, we exploit some information about the underlying representation of procedures in PLT Scheme. Specifically, we compare the contents of the closures corresponding to those functions (using simple pointer equality on the contents of the closure and the code pointer). In this case, the closure contains the free variables `bounded-bst` and `lo` and thus the closure will match any other closure that has the same value of `lo`, which suffices to avoid the redundancy seen in the example from Figure 4. Since this comparison may fail when standard compiler optimizations are performed, our implementation communicates with the compiler, telling it not to optimize these particular closures. So far, we have found this strategy for comparing contracts to be sufficiently powerful for the programs we have run. The next section discusses an experiment that demonstrates that our strategy has a significant, positive impact on the performance of our contract checker.

After some experimentation with our implementation, we discovered that a significant amount of time is spent in allocation, even with the stronger check in place. In particular, there is still significant extra allocation because the implementation allocates a record for each contract combinator. This approach becomes expensive when combined with dependent contract checking, because the allocation of the contracts happens during the traversal of the data structure. To compensate, we built a "flattening" optimization for lazy contracts that flattens nested contracts together into a single contract, in order to cut down on the allocation.

As an example, consider this contract:

```
(or/c null? (between/c 0 +∞))
```

It accepts either `null` or positive numbers. Without the optimization, the construction of this contract requires creating two records, one for the `or/c` contract and one for the `between/c` contract. With the optimization, we can simply create a single record that stores the bounds and simultaneously checks if the value is `null` or an appropriate number. Returning to Figure 5 (b), our optimization would only allocate a single record, replacing the two separate contract records with a single record for a node-between contract. Our optimization can also detect recursive contracts, so for the `bounded-bst` example, we can eliminate much of the allocation, requiring only a single allocation for each layer of the tree (to hold the new bounds).

5 Performance

This section presents the results of three experiments we performed on our implementation. Although these experiments are not conclusive, they do provide some validation of our contract checker. The first experiment validates the claims from Section 2 by showing that eagerly checking the contracts can be arbitrarily slower than lazily checking

them. The second experiment measures the cost of laziness, in the case that laziness is superfluous. The third demonstrates how our lazy contract checker behaves for more realistic applications and provides empirical evidence that it does indeed preserve the asymptotic complexity of the underlying operations.

We ran all of our experiments using PLT Scheme [13] v3.99.0.13 on a dual core 1.66 GHz Mac mini with 2 gigabytes of memory (although each test ran sequentially and only a test that disabled the stronger check allocated a significant amount of memory, discussed in Section 5.3).

5.1 The Cost of Eagerness

As we discussed earlier in this paper, the cost of eagerly checking data structure contracts can be arbitrarily bad. To verify this claim, we ran a simple test with our implementation. We built a toy program that constructs increasingly larger complete binary trees, numbers them via an inorder traversal (to satisfy the binary search tree invariant), and then measures the time it takes to search for each number.

Figure 7 shows the results. The x-axis ranges over the number of elements in the binary search trees, and the y-axis shows the slowdown as the *ratio* of the time required to call `find-bst` with the the eager contracts to the time to call `find-bst` with the lazy contracts. Each point on the graph represents a single run of each program. Even at the relatively modest size of a 10,000 element binary search tree, eager checking incurs an overhead that is more than 200 times greater than lazy checking. More worryingly, however, is the shape of the graph; as the size of the binary search tree increases, so does the factor of slowdown, meaning that eager checking is slowing down significantly more than lazy checking as the trees get bigger.

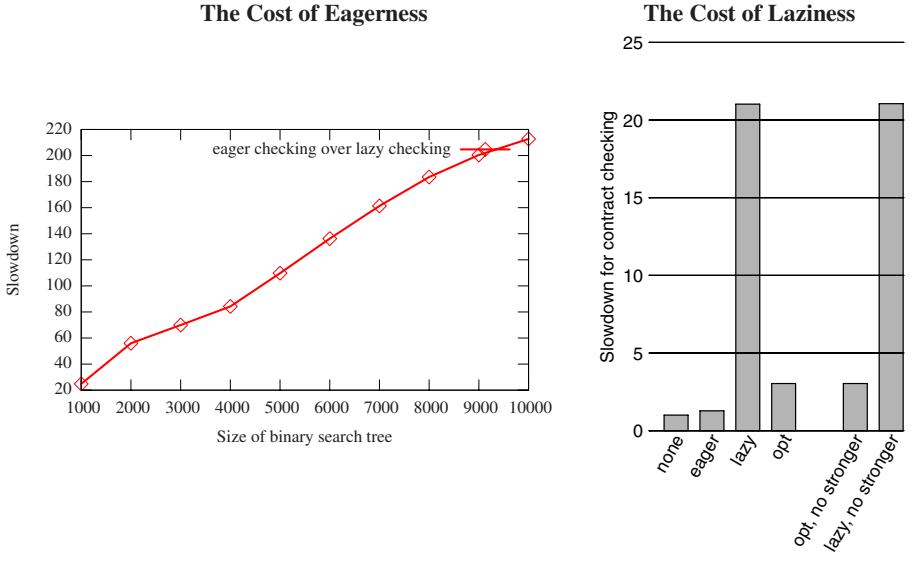


Fig. 7. Synthetic benchmark results

5.2 The Cost of Laziness

To measure the cost of laziness, we wrote a program that constructs a list of the numbers from 1 to 100,000. We did not use PLT Scheme's built-in `cons` function, because our contracts only support user-defined structs. Instead, we made a two field struct and used that for the pairs in the list. Once the list is built, the program applies different implementations of a contract that specifies that the list is sorted in ascending order, and then iterates over the list. Since the function always iterates over the entire list, delaying the contract does not improve the running time. Accordingly, this test helps us understand the cost of our implementation's bookkeeping. The right-hand side of Figure 7 shows those measurements. The height of each bar in the figure is the ratio of the performance of a particular contract to the performance of the code without any contracts.

The first four bars show the slowdown of the running time as compared to the version without contracts. The first bar (none) just gives a sense of scale; the slowdown for the version without contracts as compared to itself is 1. The second bar (eager) shows the slowdown for the eager contract that iterates down the entire list during the pre-condition checking, the third for the lazy contracts (lazy), and the fourth for lazy contracts with our flattening optimization (opt). Each bar corresponds to the average result of five runs. We see that the cost of the lazy contract bookkeeping is about a factor of 21 for this program, compared to a factor of 1.3 for the eager contract. Our optimization brings this cost down to a more reasonable factor of 3.0.

For a final experiment to measure the cost of laziness, we also set out to determine the cost of evaluating the stronger relation. For the program in this section, we know that no contract is ever going to be applied twice to the same object, so the stronger relation has no positive effect on the running time. We disabled the code that does that check and re-ran the tests. The results are shown as the final two bars in Figure 7. They show that the stronger check does not have a significant cost, when compared to the cost of the contract checking itself.

5.3 A Realistic Benchmark

For this experiment, we extracted traces of calls to a heap data structure from a colleague's vision algorithm [8]. We used four traces that are named after the images we used when extracting each trace: elephant, elephant-big, bird, and koala. The traces vary in size: elephant has roughly 22,000 inserts and 5,700 removals of minimum elements, whereas koala has more than 300,000 inserts and nearly 150,000 removals. We then coded up a binomial heap, as described in Okasaki's book [18] and ran the traces with three variations of the contracts on the heap operations: no contracts, optimized lazy contracts, and eager contracts. Accordingly, these results represent the times for only the data structure operations, not the original program that used the heap.

Figure 8(a) shows the slowdown for running the optimized lazy contract checking on heap operations; note that this chart's scale is not the same as that in Figure 7. As you can see, even though the traces vary in size, the overhead is relatively constant, encouraging us to believe that our contract checker only adds a constant overhead.

Figure 8(b) shows the slowdown for using the eagerly checked contracts on heap operations. Since these runs take a long time — running the koala trace once requires

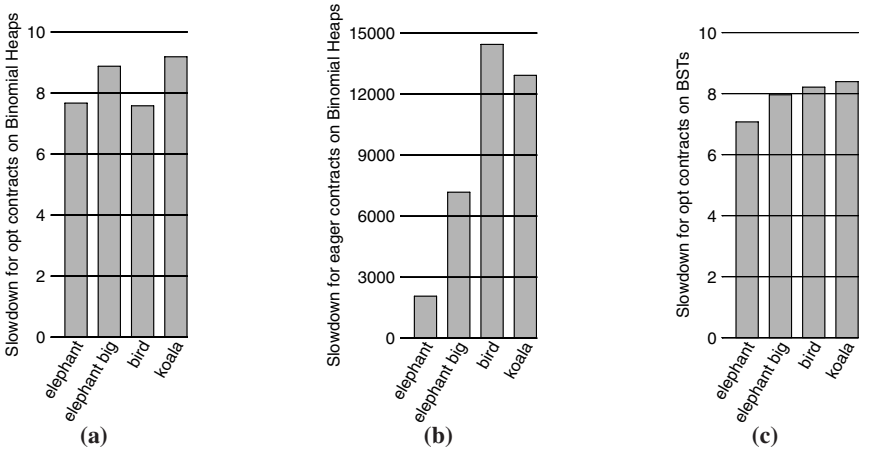


Fig. 8. Binomial heap and binary search tree experiments

about thirty CPU hours — we only ran them three times each. There are two important features of this chart. First, the scale is significantly different from that of the other two charts. The overheads are at least 2,000 and can be as bad as 14,400. Second, the overheads are not close to each other, demonstrating that the eager checking does not preserve the asymptotic complexity of the program.

We also synthesized traces for binary search trees from the heap traces. We replaced each heap insertion with a naive binary search tree insertion and replaced each extract minimum with a lookup of a random element in the tree. Figure 8(c) shows the slowdown when running the revised traces with the optimized contract checker and, as before, the overheads are relatively constant.

Finally, we performed one more experiment to test the contribution of the stronger check. We disabled the stronger check and then re-ran the optimized contract checker in the binary search tree experiment. Partway through the smallest trace, PLT Scheme had 1.5 gigabytes of resident storage (according to top) and then the machine proceeded to swap, making very little additional progress. This behavior indicates that a well-designed stronger relation is a crucial part of making the implementation practical.

6 Preserving Amortized Complexity

Implicit in the strategy of our lazy contract checker is a limitation of its expressiveness. In particular, the contract for the unexplored portion of the data structure must be expressible using only information in the explored portion of the data structure. This limitation is precisely what allows us to check the contract incrementally and to preserve the asymptotic complexity of the operations in the original program.

While this limitation still permits fairly expressive contracts, there are data structures with invariants that cannot be expressed. For example, one might wish to ensure that a binary tree is full, i.e., if the height of the tree is n, there are 2^n nodes in the tree. Intuitively, the contracts presented so far cannot express this contract because they require

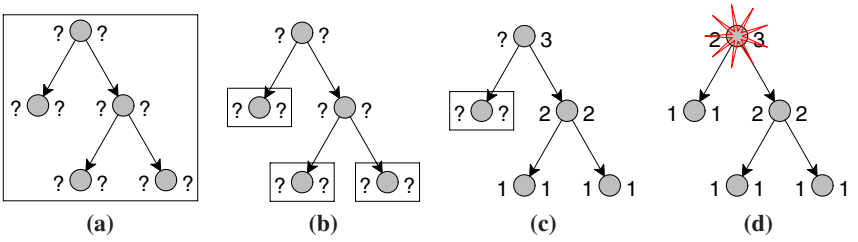


Fig. 9. Evolution of attributes during tree traversal for a full binary tree

knowledge of the particular height before reaching the leaf nodes. This restriction arises because the contracts thus far have only been based on values that are propagated "downwards", whereas fullness of a binary tree must be expressed with values that are propagated "upwards" as well. In the jargon of attribute grammars, the former are inherited attributes while the latter are synthesized attributes.

In order to check this contract, we must relax the strict constraint that contract checking will not affect the asymptotic complexity of the original program's operations. In particular, we allow the checker to preserve only the amortized asymptotic complexity of the program's operations while checking contracts that depend on the values of synthesized attributes. In order to check the full binary search tree contract, we can wait until the traversal reaches a leaf node and, at that point, propagate the height values to nodes on the path to the root.

To get a sense of how this kind of contract is checked, consider Figure 9. This contract has two synthesized attributes: a left height and a right height, denoting the height of a node's left and right children. The invariant is that both heights must be equal once they are known. Like ordinary struct contracts, contracts with attributes are checked lazily. As in Figure 3, we box the uninspected portions of the tree. Initially, each node is decorated with two question marks, indicating that the values of the left and right heights are both unknown. Figure 9(b) shows the state of the tree after inspecting the two interior nodes. At this point, none of the attribute's values are known, because none of the leaf nodes have been discovered. In Figure 9(c), the program inspects the children in the rightmost subtree. Since they have no children, their left and right height attributes are both `1`. At this point, because some attribute values have become known, propagation is triggered, resulting in the `right-height` attribute of the root becoming `3`. Finally, in Figure 9(d), the program inspects the left-most child, triggering propagation of the left height back to the root, where a contract violation is discovered, because the left height and right height of the root are not the same.

Our contract system takes care of the propagation and verification of these attributes as well as determining whether they are known or unknown. The programmer, on the other hand, must provide the logic of how those attributes are computed and what to do with them once they are known. In Figure 9, for example, our system takes care of the boxing and the propagation of the tree heights back up the tree, but it is the programmer who determines that those attributes are to be propagated upon discovering leaf nodes and that both left and right heights must be equal once they are both known. This

separation allows enough expressiveness to implement more complex contracts than our motivating example while still preserving the amortized asymptotic complexity.

Since each wrapper has a fixed c number of attributes, the propagation can occur at most c times, and each node in the tree will be inspected at most c times. Thus, the complexity of the program can only change by the constant factor, c. Because attribute evaluation may propagate an unbounded distance when just a single field is selected, however, only the amortized complexity of the original operations in the program is preserved.

7 Related Work

The idea of software contracts dates back to the 1970s [19]. In the 1980s, Meyer developed an entire philosophy of software design based on contracts, embodied in his object-oriented programming language Eiffel [16]. Nowadays, contracts are available in one form or another for many programming languages (e.g., C [23], C++ [21], Haskell [14], Java [15], Perl [5], Python [20], Scheme [22], and Smalltalk [1]).

Although the authors did not make the connection until much of this work had been done, this work is a direct intellectual descendent of Okasaki's dissertation [17], where Okasaki demonstrates that a controlled amount of laziness, in an otherwise strict language, makes achieving desired asymptotic bounds tractable. We cannot, however, use Okasaki's `$` operator directly, because we need fine-grained control over the laziness to exploit the stronger relation.

From a contracts perspective, our work is anticipated by Chitil, McNeill, and Runciman's and Chitil and Huch's Lazy Assertions [2,3,4]. They observe that eagerly checking assertions in a lazy setting can introduce non-termination where none should rightly be. In particular, a strict assertion on an infinite list should not explore the entire list unless the program itself explores the entire list. They attempt to preserve laziness in a lazy world, whereas our work attempts to add laziness to a strict world. Despite starting from very different foundations, both arrive at the conclusion that laziness for checking contracts on data structures is necessary. From a technical point of view, we believe that the stronger relation should carry back to their setting and should help them with memory use, and that the ideas in Section 6 should also apply to their system.

Hinze, Jeuring, and Löh's contract checker [14] is also a contract checker for Haskell (that correctly handles blame), but their checker explores parts of the data structure that the program does not. For example, the `is(sort)` example contract in Section 6 of their paper explores the entire list; a similar contract in our system would not.

Beyond that, there is little other work on checking data structure contracts, except when using naive strategies. Eiffel, the language most focused on contract checking, provides no native support for lazy contract checking. Tremblay and Cheston [24] wrote an algorithms and data structures textbook using Eiffel, but the contracts in their text either only partially check the data structure invariants or check them as the data structure is constructed.

8 What Have We Gained?

In some sense, this work puts data structure contract checking on an even footing with function-based contract checking. Specifically, when checking a contract on a function,

violations can go undetected if the function is never called with an input that would trigger an error. Similarly, consider this (supposed) binary search tree:

```
(make-node 5
           (make-node 7 null null)
           (make-node 207 null null))
```

If `find-bst` is called with that tree and, say, `6`, the contract checker will not discover the violation. Even worse, if it is called with `7`, `find-bst` will indicate that `7` is not in the binary search tree, and the contract checker will still fail to detect the violation. Of course, similar behavior can happen with functions (in fact, this binary search tree could be encoded as a function to achieve precisely the same behavior) and yet function contracts enjoy wide-spread use.

We believe that our data structure contracts have the potential to enjoy similar wide-spread use, for two reasons. First, it is rare for a data structure to be built that will not eventually be completely explored in a long-running application. Even though the two calls to `find-bst` above do not detect the violation, it seems likely that some later call to `find-bst` will ask for a number smaller than `5`, resulting in a contract violation.

Second, our checker makes checking data structure contracts feasible. As discussed in Section 5.3, using either the naive strategy of eagerly checking the contracts, or even avoiding the stronger check makes checking the contracts infeasible, for at least one realistic program. Intuitively, we expect the naive strategy to fail in general, simply because the change to the asymptotic complexity incurred by the naive checker is a tremendous expense.

Fundamentally, the question we ask is how much contract checking can we expect a program to be able to afford? Our contract checker represents one answer to this question that does not take into account any a priori knowledge about the program's behavior; it provides a maximal amount of contract checking that we can reasonably expect the program to be able to afford, namely a constant factor.

Acknowledgements. Thanks to Ryan Culpepper and Matthew Flatt for PLT Scheme infrastructure support and to Matthew for comments on drafts of the paper. Thanks to Pedro Felzenszwalb for supplying us with the heap traces we use in our experiments. Also, thanks to Matthias Felleisen, Simon Peyton Jones, and Jacob Matthews for several enlightening discussions and comments on this paper. This work is supported in part by the NSF.

References

1. Carrillo-Castellon, M., Garcia-Molina, J., Pimentel, E., Repiso, I.: Design by contract in Smalltalk. Journal of Object-Oriented Programming 7(9), 23–28 (1996)
2. Chitil, O., Huch, F.: A pattern logic for prompt lazy assertions. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 126–144. Springer, Heidelberg (2007)
3. Chitil, O., Huch, F.: Monadic prompt lazy assertions in Haskell. In: Asian Symposium on Programming Languages and Systems (2007)
4. Chitil, O., McNeill, D., Runciman, C.: Lazy assertions. In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, pp. 1–19. Springer, Heidelberg (2004)

5. Conway, D., Goebel, C.G.: Class: Contract – design-by-contract OO in Perl, http://search.cpan.org/~ggoebel/Class-Contract-1.14/
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1990)
7. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs. MIT Press, Cambridge (2001), http://www.htdp.org/
8. Felzenszwalb, P., McAllester, D.: A min-cover approach for finding salient curves. In: IEEE Workshop on Perceptual Organization in Computer Vision (2006), http://people.cs.uchicago.edu/~pff/papers/
9. Findler, Barzilay, Blume, Codik, Felleisen, Flatt, Huang, Matthews, McCarthy, Scott, Press, Rainey, Reppy, Riehl, Spiro, Tucker, Wick: In: The eighth annual ICFP programming contest, http://icfpc.plt-scheme.org/
10. Findler, R.B., Blume, M.: Contracts as pairs of projections. In: International Symposium on Functional and Logic Programming, pp. 226–241 (2006)
11. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of ACM SIGPLAN International Conference on Functional Programming, pp. 48–59 (2002)
12. Flanagan, C., Sabry, A., Duba, B., Felleisen, M.: The essence of compiling with continuations. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (1993)
13. Flatt, M.: PLT MzScheme: Language manual. Technical Report PLT- TR05-1-v300, PLT Scheme Inc. (2005), http://www.plt-scheme.org/techreports/
14. Hinze, R., Jeuring, J., Löh, A.: Typed contracts for functional programming. In: International Symposium on Functional and Logic Programming (2006)
15. Karaorman, M., Hölzle, U., Bruno, J.: jContractor: A reflective Java library to support design by contract. In: Cointe, P. (ed.) Reflection 1999. LNCS, vol. 1616. Springer, Heidelberg (1999)
16. Meyer, B.: Eiffel: The Language. Prentice Hall, Englewood Cliffs (1992)
17. Okasaki, C.: Purely Functional Data Structures. PhD thesis, Carnegie Mellon University, Technical Report CMU-CS-96-177 (September 1996)
18. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1999)
19. Parnas, D.L.: A technique for software module specification with examples. Communications of the ACM 15(5), 330–336 (1972)
20. Plösch, R.: Design by contract for Python. In: IEEE Proceedings of the Joint Asia Pacific Software Engineering Conference (1997), http://citeseer.nj.nec.com/257710.html
21. Plösch, R., Pichler, J.: Contracts: From analysis to C++ implementation. In: Technology of Object-Oriented Languages and Systems, pp. 248–257 (1999)
22. PLT. PLT MzLib: Libraries manual. Technical Report PLT-TR2007-4-v372, PLT Scheme Inc. (2007), http://www.plt-scheme.org/techreports/
23. Rosenblum, D.S.: A practical approach to programming with assertions. IEEE Transactions on Software Engineering 21(1), 19–31 (1995)
24. Tremblay, J.-P., Chesterton, G.A.: Data Structures and Software Development in an Object-Oriented Domain: Eiffel Edition. Prentice Hall, Englewood Cliffs (2001)