CoCoA-5 - Slug #1228

SLUG: filling an array

30 Sep 2018 10:08 - John Abbott

Status:	In Progress	**Start date:**	30 Sep 2018
Priority:	High	**Due date:**	
Assignee:	John Abbott	**% Done:**	30%
Category:	bug	**Estimated time:**	0.00 hour
Target version:	CoCoA-5.4.2	**Spent time:**	3.55 hours

Description

It seems that assigning to an array element is surprisingly slow.

Look at the timings in the following session:

```
>>> t0 := CpuTime();
>>> c := 0; for J := 1 to N/2 do if gcd(J,N) = 1 then incr(ref c); endif endfor;
>>> TimeFrom(t0);
8.734
>>> t0 := CpuTime();
>>> d := 0; for J := 1 to N/2 do if gcd(J,N) = 1 then d := d+mod(J^2,N); endif endfor;
>>> TimeFrom(t0);
7.442
>>> sieve := NewList(N);
>>> t0 := CpuTime();
>>> for J := 1 to N/(17*38) do if gcd(J,N) = 1 then sieve[mod(J^2,N)]:=1; endif endfor;
>>> TimeFrom(t0);
42.705
```

There are 3 loops. Why is the first loop SLOWER than the second one??
Why is the third loop FAR SLOWER than either the first or second ones?
(note that the third loop does only about 0.3% of the iterations of the other loops)

Related issues:

Related to CoCoA-5 - Slug #1229: append is slow	**New**	**05 Oct 2018**
Related to CoCoA-5 - Slug #96: sort is too slow	**New**	**01 Mar 2012**
Related to CoCoA-5 - Slug #862: append has bad complexity	**New**	**05 Apr 2016**
Related to CoCoA-5 - Slug #31: theValue makes copy	**In Progress**	**15 Nov 2011**

History

#1 - 30 Sep 2018 10:11 - John Abbott

- Priority changed from Normal to High

I forgot to mention that N := 9699690 (product of all primes up to 20).

The third loop is **APPALLINGLY SLOW**, **TRULY EMBARRASSINGLY SLOW**.

I have a nasty fear that sieve[...] := 1; actually makes a copy of the entire list sieve (which contains nearly 10 million elements).

I do hope that this is easy to fix.

#2 - 30 Sep 2018 10:19 - John Abbott

- Description updated

#3 - 01 Oct 2018 11:16 - John Abbott

- Status changed from New to In Progress

- % Done changed from 0 to 10

I have just tried the third loop again, but after making the list sieve twice as long, and the loop takes almost twice as long. So the assignment sieve[...] := 1 has cost proportional to the length of the list sieve.

Ouch!

#4 - 02 Oct 2018 15:20 - John Abbott

- % Done changed from 10 to 20

I have used profiling to confirm that the list is indeed copied (but why???)

The test code is:

```
LongList := NewList(1000000);

t0 := CpuTime();
for j := 1 to 100 do
  LongList[j] := j;
endfor;
println "Loop time: ", TimeFrom(t0);
```

Profiler reports:

```
                0.23    1.05     100/100         CoCoA::InterpreterNS::LeftValue::obtainOwnership() [6]
[7]     83.7    0.23    1.05     100         CoCoA::InterpreterNS::LIST::needsToBeCopiedBeforeChanges() const 
[7]
                0.23    0.07 100000100/100000100     bool __gnu_cxx::operator!=<boost::intrusive_ptr<CoCoA::In
terpreterNS::RightValue> const*, .... > const&) [8]
```

In Interpreter.C around line 2355 the source says:

```
bool MAT::needsToBeCopiedBeforeChanges() const {
    return this->getRefCount()>1;
}
```

So the question is: why is the ref count greater than 1???

#5 - 02 Oct 2018 15:26 - John Abbott

In Interpreter.C around line 2065 there is:

```
bool IndexedAccess::assignmentNeedsOwnership() const {
    return true;
}
```

In Interpreter.C around line 4210 there is:

```
void AssignmentStatement::implExecute(RuntimeEnvironment *runtimeEnv) {
    const intrusive_ptr<LeftValue> leftValue = this->leftExp->evalAs<LeftValue>(runtimeEnv);
    runtimeEnv->interpreter->checkForInterrupts(this->leftExp);
    intrusive_ptr<RightValue> rightValue = this->rightExp->evalAs<RightValue>(runtimeEnv);
    if (leftValue->assignmentNeedsOwnership())
        leftValue->obtainOwnership();
    leftValue->assign(rightValue, this->rightExp->getBegin(), this->rightExp->getEnd(), runtimeEnv);
}
```

#6 - 02 Oct 2018 15:46 - John Abbott

- Assignee set to John Abbott

- % Done changed from 20 to 30

I have changed IndexedAccess::assignmentNeedsOwnership so that it gives false.
The loop is much faster (0.001s against 11s), and the CoCoA5 tests all pass.

So, is it safe to make that change???

The slow loop in the main description now takes 0.3s (with debugging and profiling on) against 42s (optimized).

I'll try asking Giovanni...

#7 - 02 Oct 2018 16:51 - John Abbott

Now I fear there may be some "strange" situations in which nasty things may occur:

```
define HEAD(ref L) L := L[1]; return L; enddefine;
```

```
L := [1,2];
L[2] := HEAD(ref L); --> asking for trouble!
```

With my modified CoCoA-5 this produces:

```
--> ERROR: Expecting type LIST, MAT or MATRIXROW, but found type INT
--> L[2] := HEAD(L); --> asking for ...
--> ^
```

Not so helpful, but I had feared a crash or memory corruption.

Maybe the semantics of an assignment of the form L[i] := blah should be:

1. evaluate blah
2. check that L is indexable and that i is a valid index; if so, then assign.

Not sure what the actual semantics are...

#8 - 02 Oct 2018 17:05 - John Abbott

A potential disadvantage of my proposed semantics is that if the RHS is expensive to evaluate, and the LHS is "obviously wrong" then the user must wait for the evaluation of RHS to finish before discovering the error.

What should happen with this input?

```
>>> J := 1;
>>> define MakeRecord(ref A) A := record[]; return 1; enddefine;
>>> J.xyz := MakeRecord(J);
```

Currently it complains:

```
--> ERROR: Expecting type RECORD, but found type INT
--> J.xyz := MakeRecord(J);
--> ^
```

I am inclined to think that a user who writes an assignment LHS := RHS; where evaluating RHS may change the structure of LHS should accept whatever CoCoA-5 chooses to do... but CoCoA-5 should not SEGV, or corrupt memory or do anything else "nasty": it should either do something "legal" or return a proper CoCoA error.

#9 - 02 Oct 2018 19:43 - Anna Maria Bigatti

John Abbott wrote:

> I am inclined to think that a user who writes an assignment LHS := RHS; where evaluating RHS may change the structure of LHS should accept whatever CoCoA-5 chooses to do...

Michael Abshoff used to say "you can't prevent the user to shoot himself in the foot".

PS with my old CoCoA I get the same errors you wrote.

#10 - 03 Oct 2018 14:27 - John Abbott

Yes, we cannot (and probably should not) stop the user from "shooting himself in the foot".
But CoCoA-5 should never crash (SEGV, or other naughty memory access).

#11 - 03 Oct 2018 14:43 - John Abbott

Here is an alternative to my comment about the semantics of assignment (in comment 7):

1. check that LHS is writable (valid index, etc)
2. evaluate RHS
3. check **again** that LHS is writable, and then write the value.

I would hope that checking writability is cheap...

#12 - 03 Oct 2018 15:03 - Anna Maria Bigatti

John Abbott wrote:

> I would hope that checking writability is cheap...

We will make some checks.
I want to design some speed tests mechanism.
(Cannot really be based on the current test examples because they are for testing correctedness, and effectively expansible at any time)

#13 - 04 Oct 2018 09:01 - John Abbott

My proposal in comment 11 is **bad** because, for instance, the computation of an index may be lengthy or may have side-effects. The user would be surprised to find the index computed twice.

Now I propose:

1. check LHS, and cache the index values (if any)
2. evaluate RHS
3. check existence/writability of LHS using the cached indexes

It seems that the interpreter may already do something similar to this, because my tests gave errors (but the error messages were not very helpful...)

NOTE a naughty user could write something like L[expr1] := expr2 where expr1 has a side-effect changing L, and expr2 has a side-effect changing the way expr1 is evaluated. Hopefully Giovanni had already thought of this!

#14 - 05 Oct 2018 17:43 - John Abbott

It does not really matter how we evaluate LHS := RHS.
Either side could be expensive to evaluate. Publicly we should say that the order of evaluation is not rigidly specified -- this lets us change it in the future. Anyway, it is rather risky writing code which relies on order of evaluation inside a single expression.

Anyway, my suggestion in comment 13 is reasonable (and seems to be what the interpreter actually does, based on a few tests).

#15 - 05 Oct 2018 17:44 - John Abbott

- Related to Slug #1229: append is slow added

#16 - 29 Apr 2019 14:51 - John Abbott

- Related to Slug #96: sort is too slow added

#17 - 29 Apr 2019 15:49 - John Abbott

- Related to Slug #862: append has bad complexity added

#18 - 29 Apr 2019 18:00 - John Abbott

I think I have found a situation where my simple idea in comment 6 causes trouble...

```
/**/ L := [2,3,1];
/**/ sorted(L);
[1,2,3];
/**/ L;
[1,2,3]; // OUCH!!??!!
```

Boohoo!!

Note: JAA has no idea how to fix this :-(But it is a terrible SLUG.

#19 - 01 Oct 2019 14:29 - John Abbott

- Target version changed from CoCoA-5.3.0 to CoCoA-5.4.0

#20 - 28 May 2021 20:37 - John Abbott

- Related to Slug #31: theValue makes copy added

#21 - 03 Feb 2022 19:42 - John Abbott

- Target version changed from CoCoA-5.4.0 to CoCoA-5.4.2