goblint-zarith

A fork of Zarith that is more compatible with Js_of_ocaml

Cakirer, Kerem

July 12, 2021

1. Introduction

This document aims to explain why I maintain a fork of Zarith for the Goblint project and the changes I've made to Zarith. Hopefully, it will aid in further maintenance of this fork in case I stop working on Gobview.

1.1. Rationale

For someone who has a C or Java background, integers in both OCaml and JavaScript have unintuitive behavior. In OCaml, complex objects tend to get allocated on the heap, and a variable is usually nothing more than a pointer. Since "int" is one of the most frequently used types, OCaml employs a trick to avoid the heap allocation of "int"s. It reserves the least significant bit of an "int" as a tag. This optimization works because, thanks to alignment requirements on most architectures, the LSB of a pointer is always zero. As a result, the OCaml runtime can check this bit to differentiate between pointers to heap-allocated objects and "int"s (which have "1" as the LSB). However, as a side effect, OCaml "int"s are 31-bit and 63-bit on 32-bit and 64-bit systems, respectively.

JavaScript integers are also unintuitive in a unique way: They do not exist. All numeric values in JavaScript are IEEE 754 double-precision floating-point numbers. This quirk practically limits the width of an "integer" in JavaScript to 53-bits. Js_of_ocaml imposes a further restriction on integer sizes by pretending to be a 32-bit system.

Goblint uses Zarith to represent arbitrary-precision integers. Behind the scenes, Zarith uses the GNU MP library to represent big integers. However, just like OCaml, Zarith implements a tagging system to optimize smaller integers. Zarith uses the native OCaml "int" to store all integers that fit in an "int" and checks the LSB to differentiate between small and big integers. At runtime, it will dynamically switch between the two representations as necessary.

Since GMP and most of Zarith are in C, they don't work in a web browser. The "zarith_stubs_js" package reimplements the native parts of Zarith in JavaScript to overcome this limitation. This solution works provided that the two worlds do not cross. However, Gobview aims to reuse the serialized state of the native Goblint in the browser,

and, as a result, this limitation becomes an issue. The problem is that regular OCaml "int"s and (small) Zarith integers are only distinct during compile-time. The OCaml runtime cannot distinguish between "int"s and small Zarith integers since they share the same format. As a result, the OCaml runtime will serialize a small Zarith integer the same way it would serialize an actual "int". Unfortunately, because Js_of_ocaml emulates a 32-bit system, Js_of_ocaml will refuse to deserialize some "small" integers (e.g., anything equal to or larger than 2^{30}).

2. Solutions

2.1. Disabling optimizations

This solution is the easiest and the least intrusive to implement. The idea behind it is simple: We disable the small "int" optimization and allocate integers on the heap with GMP, no matter what size they are. Unfortunately, this has a non-negligible impact on performance. (see Table 1) Therefore, it makes sense to at least try out other alternatives, but it is always possible to fall back onto this solution if the others don't pan out or become too difficult to maintain in the future.

Program	LoC	With opt.	Without opt.	Impact
401.bzip2	8090	21.58 s	23.10 s	7%
429.mcf	1608	1.33 s	1.37 s	3%
433.milc	12274	46.44 s s	48.84 s	5%
456.hmmer	33776	50.04 s	53.07 s	6%
figlet-2.2.5	7902	15.36 s	16.40 s	7%
maradns-1.4.06_zoneserver	32034	95.97 s	104.86 s	9%

Table 1: Benchmarking single-threaded programs. See Appendix A for more details.

2.2. Patching Js_of_ocaml

Yet another solution is to fork Js_of_ocaml to use "BigInt"s instead of floating-point numbers for "int"s. It would be possible to accomplish this by patching the code generation module of Js_of_ocaml and some of the primitive functions defined as part of the runtime. The main advantage of this solution is that Goblint itself needs absolutely no modifications for it to work. Unfortunately, the disadvantages dominate. First of all, Js_of_ocaml is a much more complicated project than Zarith, and a fork of it is likely to incur higher maintenance costs. In addition, the decision to represent "int"s as floating-point numbers is not a minor implementation detail, and some libraries like zarith_stubs_js implicitly depend on it. As a result, forking the Js_of_ocaml compiler is not enough on its own, which makes this solution too costly.

2.3. Partially disabling optimizations

Partially disabling optimizations is the solution I've decided to implement. To summarize, I patch the native version of Zarith to use a more restricted range for what it considers to be small integers and make it heap allocate for integers that wouldn't fit in this range.

The first idea I had was to make Zarith emulate a 32-bit system, even on 64-bit systems. This idea did not pan out because the performance was still too similar to the version without the small integer optimization. (see Table 2) It is not clear why this is the case, but I suspect "INT_MIN" & "INT_MAX" to be the main culprits. These two values occur in cases where the static analysis fails to yield a meaningful approximation, and they will not fit in a 31-bit integer that a 32-bit build of Zarith would use.

The real solution is a bit more involved. Instead of emulating a 32-bit system, we emulate 53-bits. This way, we end up with a performance profile that matches the native version, (see Table 2 and 3) and everything still works in the browser. Because Js_of_ocaml considers such integers to be too large to deserialize, this change requires a patch on the Js_of_ocaml side as well. However, it is possible to override Js_of_ocaml primitives without forking the project, so this does not pose a significant challenge.

Program	Native	53-bit	32-bit	Without opt.
401.bzip2	21.58 s	21.52 s	22.86 s	23.10 s
429.mcf	1.33 s	1.33 s	1.36 s	1.37 s
433.milc	46.44 s	46.57 s	48.08 s	48.84 s
456.hmmer	50.04 s	50.23 s	51.53 s	53.07 s
figlet-2.2.5	15.36 s	15.47 s	16.24 s	16.40 s
maradns-1.4.06_zoneserver	95.97 s	96.04 s	102.91 s	104.86 s

Table 2: Benchmarking single-threaded programs. See Appendix A for more details.

Program	Privatization	Native	53-bit
ypbind	lock	19.33 s	19.39 s
ypbind	write	22.69 s	22.83 s
ypbind	write+lock	25.82 s	25.87 s
marvell1	lock	33.09 s	33.11 s
marvell1	write	45.58 s	45.30 s
marvell1	write+lock	53.37 s	53.36 s

Table 3: Benchmarking the tracing feature of Goblint. See Appendix A for more details.

I prefer this solution because the Zarith codebase is small and clean enough, and its invariants are easy to keep in consideration. Also, it is possible to use QCheck to compare the patched version of Zarith with the original implementation as an additional step to ensure correctness, something that is hard to accomplish with a patched version of Js_of_ocaml.

3. Implementation

3.1. Assumptions & Constraints

Zarith helpfully lists its program invariants in "caml_z.c." (see Listing 1) The first one ("if the number fits in an int, it is stored in an int, not a block") is important because Zarith consequently assumes that two different representations ("int" and block) can never have the same value. "equal", in particular, depends on this and will break (i.e., return false even if the parameters are equal) when this invariant gets violated. (see Listing 2) As a result, the patches to Zarith mustn't cause any inconsistency in this regard.

```
120 /*
121   A z object x can be:
122   - either an ocaml int
123   - or a block with abstract or custom tag and containing:
124     . a 1 value header containing the sign Z_SIGN(x) and the size Z_SIZE(x)
125     . Z_SIZE(x) mp_limb_t
126
127   Invariant:
128   - if the number fits in an int, it is stored in an int, not a block
129   - if the number is stored in a block, then Z_SIZE(x) >= 1 and
130   the most significant limb Z_LIMB(x)[Z_SIZE(x)] is not 0
131  */
```

Listing 1: Zarith's invariants

```
1116   /* Value-equal small integers are equal.
1117      Pointer-equal big integers are equal as well. */
1118   if (arg1 == arg2) return Val_true;
1119   /* If both arg1 and arg2 are small integers but failed the equality
1120      test above, they are different.
1121      If one of arg1/arg2 is a small integer and the other is a big integer,
1122      they are different: one is in the range [Z_MIN_INT,Z_MAX_INT]
1123      and the other is outside this range. */
1124   if (Is_long(arg2) || Is_long(arg1)) return Val_false;
```

Listing 2: The first few lines of "ml_z_equal".

In addition to this, we also have our own assumptions. For the patches, we assume the host system is 64-bit because on a 32-bit system, everything already works as expected, and no patches are necessary. Furthermore, since this fork has to work with "zarith_stubs_js", we cannot introduce any new primitives.

3.2. Constants

In "caml_z.c", various constants and macros are defined to avoid code duplication. For example, the "Z_FITS_INT" macro uses "Z_MAX_INT" and "Z_MIN_INT" to see if a native integer would fit in an OCaml "int". (see Listing 3) Since we simulate a 53-bit system, we redefine "Z_MAX_INT" as "0x7fffffffffff" ($= 2^{51} - 1$) and "Z_MIN_INT" as "-0x8000000000000" ($= -2^{51}$). [1]

```
161 #define Z_FITS_INT(v)  ((v) >= Z_MIN_INT && (v) <= Z_MAX_INT)
```

Listing 3: The definition of "Z_FITS_INT"

Note that blindly modifying constants is not always the correct choice. For example, Zarith uses "Z_HI_INT" and similarly named constants to check if it can convert its integers back to OCaml integer types. (see Listing 4) Redefining these wouldn't necessarily break Zarith, but it leads to spurious unit test failures; therefore, it is still beneficial to keep the old behavior.

```
183 /* hi bit of OCaml int32, int64 & nativeint */
184 #define Z_HI_INT32   0x80000000
185 #define Z_HI_INT64   0x8000000000000000LL
186 #ifdef ARCH_SIXTYFOUR
187 #define Z_HI_INTNAT  Z_HI_INT64
188 #define Z_HI_INT     0x4000000000000000
```

Listing 4: Zarith uses these for "to_int", "to_int32", etc.

3.3. ml_z_reduce

Zarith uses "ml_z_reduce" for normalizing "big" integers. While we don't modify this function in any way, it is necessary to understand how it works to ensure the correctness of our patch set.

A "big" integer in Zarith is equivalent to an array of word-size integers (also called "limbs") plus a header part that contains the size of the array as well as the sign of the number. "ml_z_reduce" removes any null blocks that appear at the end of the array and readjusts the size if necessary. If no blocks remain at the end of this process, it returns a zero. If only a single block remains, and the value of the block resides in the range [Z_MIN_INT, Z_MAX_INT], "ml_z_reduce" converts it to a "small" integer.

Zarith already uses "ml_z_reduce" quite intensively, so most functions don't need any modifications at all. It is mainly the conversion functions that need fixing.

[1] Since floating-point numbers use a sign bit instead of two's complement, we can make the maximum as large as 2^{52}. I don't, because Zarith might implicitly expect this constant to be defined as $2^{n-2} - 1$, where "n" is the word size.

3.4. of_int

Zarith implements "of_int" as a simple identity transformation. (see Listing 5) We redefine "of_int" as a proper function, and in cases where the input would fit in a 52-bit integer, we still perform an identity transformation. However, when it doesn't, we use the "ml_z_of_int" function from "caml_z.c" to create an arbitrary-precision integer.

```
30 external of_int: int -> t = "%identity"
```

Listing 5: Definition of "of_int"

Zarith already provides "ml_z_of_int", but we have to disable the "Z_USE_NATINT" branch for it to work correctly because otherwise, it will just return the input parameter. (see Listing 6)

```
441 CAMLprim value ml_z_of_int(value v)
442 {
443 #if Z_USE_NATINT
444   Z_MARK_OP;
445   return v;
446 #else
447   intnat x;
448   value r;
449   Z_MARK_OP;
450   Z_MARK_SLOW;
451   x = Long_val(v);
452   r =  ml_z_alloc(1);
453   if (x > 0) { Z_HEAD(r) = 1; Z_LIMB(r)[0] = x; }
454   else if (x < 0) { Z_HEAD(r) = 1 | Z_SIGN_MASK; Z_LIMB(r)[0] = -x; }
455   else Z_HEAD(r) = 0;
456   Z_CHECK(r);
457   return r;
458 #endif
459 }
```

Listing 6: Definition of "ml_z_of_int". By default, it just returns the input parameter.

3.5. of_float

Zarith implements "of_float" via the "ml_z_of_float" function in C. As it stands, this function has two issues.

The first one is quite simple: "ml_z_of_float" uses "Z_MAX_INT_FL" and "Z_MIN_INT_FL" to decide whether an OCaml "float" fits in an "int", and these constants are too large for our 52-bit integers. (see Listing 7) We redefine their values as "Z_MAX_INT" and

"Z_MIN_INT", respectively, because a floating-point number in this range will always fit in a 52-bit integer.

```
533 CAMLprim value ml_z_of_float(value v)
534 {
535   double x;
536   int exp;
537   int64_t y, m;
538   value r;
539   Z_MARK_OP;
540   x = Double_val(v);
541 #if Z_USE_NATINT
542   if (x >= Z_MIN_INT_FL && x <= Z_MAX_INT_FL) return Val_long((intnat) x);
```

Listing 7: Snippet from "ml_z_of_float"

The other issue here is the 64-bit code branch. (see Listing 8) The mantissa of the input may end up being too large for a 52-bit integer, so we have to allocate a large integer, just like in the 32-bit branch. However, in the 64-bit version, we only need a single "limb" because the "limbs" of an GMP integer adhere to the CPU word size. As a result, copying the 32-bit version would end up enlarging the mantissa on 64-bit systems.

```
550   exp = ((y >> 52) & 0x7ff) - 1023; /* exponent */
551   if (exp < 0) return(Val_long(0));
552   if (exp == 1024) ml_z_raise_overflow();  /* NaN or infinity */
553   m = (y & 0x000fffffffffffffLL) | 0x0010000000000000LL; /* mantissa */
554   if (exp <= 52) {
555     m >>= 52-exp;
556 #ifdef ARCH_SIXTYFOUR
557     r = Val_long((x >= 0.) ? m : -m);
558 #else
559     r = ml_z_alloc(2);
560     Z_LIMB(r)[0] = m;
561     Z_LIMB(r)[1] = m >> 32;
562     r = ml_z_reduce(r, 2, (x >= 0.) ? 0 : Z_SIGN_MASK);
563 #endif
```

Listing 8: Another snippet from "ml_z_of_float". Notice how the 32-bit version splits the mantissa and stores each part in a separate limb.

3.6. of_substring_base

"of_substring_base" creates a Zarith integer from a (sub)string. Functions like "of_string", "of_substring", and "of_string_base" all use "of_substring_base" underneath. Internally, this function uses the "Z_BASEXX_LENGTH_OP" family of constants to decide whether the input will fit in a native integer. Luckily, the comments explain how the values of the constants were selected, so we can easily repeat the same process to redefine them. (see Listing 9)

```
194 /* safe bounds for the length of a base n string fitting in a native
195    int. Defined as the result of (n - 2) log_base(2) with n = 64 or
196    32.
197 */
198 #ifdef ARCH_SIXTYFOUR
199 #define Z_BASE16_LENGTH_OP 15
200 #define Z_BASE10_LENGTH_OP 18
201 #define Z_BASE8_LENGTH_OP 20
202 #define Z_BASE2_LENGTH_OP 62
```

Listing 9: Definition of "Z_BASEXX_LENGTH_OP" constants.

3.7. succ, pred, abs & neg

These functions all use either "min_int" or "max_int" to do a boundary check on the input and decide how to proceed. (see Listing 10) As a result, they sometimes create "int"s that do not fit in a 52-bit integer. The solution is to redefine "min_int" and "max_int."

```
119 let succ x =
120   if is_small_int x && unsafe_to_int x <> max_int
121   then of_int (unsafe_to_int x + 1)
122   else c_succ x
```

Listing 10: Definition of "succ". The others look similar.

3.8. add & sub

For these operations, unless an input parameter is not a small integer, Zarith first computes the result and then performs some bit-level hackery to check whether it has overflowed or not. (see Listing 11) As expected, this doesn't work as intended in our case. As expected, this doesn't work as intended in our case. To avoid this issue, we patch "add" and "sub" to always call "c_add" and "c_sub", respectively. Additionally, since we call the C primitives directly now, we have to modify them to do "fast path" checks. (see Listing 12)

```
41 let add x y =
42   if is_small_int x && is_small_int y then begin
43     let z = unsafe_to_int x + unsafe_to_int y in
44     (* Overflow check -- Hacker's Delight, section 2.12 *)
45     if (z lxor unsafe_to_int x) land (z lxor unsafe_to_int y) >= 0
46     then of_int z
47     else c_add x y
48   end else
49     c_add x y
```

Listing 11: Definition of "add". "c_add" is a binding for "ml_z_add".

```
1404 CAMLprim value ml_z_add(value arg1, value arg2)
1405 {
1406   Z_MARK_OP;
1407   Z_CHECK(arg1); Z_CHECK(arg2);
1408 #if Z_FAST_PATH && !Z_FAST_PATH_IN_OCAML
1409   if (Is_long(arg1) && Is_long(arg2)) {
1410     /* fast path */
1411     intnat a1 = Long_val(arg1);
1412     intnat a2 = Long_val(arg2);
1413     intnat v = a1 + a2;
1414     if (Z_FITS_INT(v)) return Val_long(v);
```

Listing 12: Partial definition of "ml_z_add". By default, "Z_FAST_PATH_IN_OCAML" is true, and the "fast path" is disabled.

3.9. mul

For small inputs, "mul" uses the C function "ml_z_mul_overflows" to detect overflows. (see Listing 13) The OCaml part of "mul" itself doesn't need any modifications; however, the helper function "ml_z_mul_overflows" must be corrected to use "Z_FITS_INT".

An additional challenge here is that "ml_z_mul_overflows" provides multiple implementations of the same functionality for cross-platform portability. (see Listing 14) For now, I only fix the first code path because it uses the GCC/Clang intrinsic "__builtin_mul_overflow" and already covers all the platforms we actively use/target.

4. Distribution

Yet another issue is how we distribute this fork of Zarith and integrate it with Goblint. Although my patches don't have a significant impact on performance, there might be undiscovered correctness issues. Therefore, I tried to find a solution that satisfies the following criteria:

```
67 let mul x y =
68   if is_small_int x && is_small_int y
69   && not (mul_overflows (unsafe_to_int x) (unsafe_to_int y))
70   then of_int (unsafe_to_int x * unsafe_to_int y)
71   else c_mul x y
```

Listing 13: Definition of "mul"

- Goblint should ideally only use this fork when the user also wishes to build Gobview.
- There shouldn't be any complex changes in Goblint itself.
- It should be hard to create a misconfigured build of Goblint. (Unpatched Zarith + Gobview)

The first solution that comes to mind is to provide our fork of Zarith as a separate package (e.g., "gobzarith"), create a wrapper module for Zarith, and use the "alternative dependencies" feature of Dune to select the correct implementation of Zarith. Unfortunately, this doesn't work because we also want any dependencies of Goblint to use our fork.

To avoid a scenario where the dependencies of Goblint and Goblint itself use different versions of Zarith, we pretty much have to use pinned packages. However, pinning has its own set of shortcomings. Opam supports conditional dependencies, but it has no notion of "conditional pinnings." As a result, the user has to manage the pin for Zarith manually. They have to remember to remove it if they don't need Gobview and add it back if they change their mind.

To avoid the aforementioned issues, I decided to handle these details in the forked package itself. Instead of modifying Zarith's source files directly, I develop the patchset in a separate branch and distribute them as a patch file in a release branch. Then, if Js_of_ocaml appears to be installed on the system, Opam applies these changes before the build process. Otherwise, the original version is built. I also made Js_of_ocaml an optional dependency of Zarith to trigger a rebuild of Zarith and its dependents, even if the user installs Js_of_ocaml at a later point. This solution satisfies my criteria and is transparent to Goblint.

5. Testing

Finally, we have to consider testing. Zarith already provides a reasonably robust test suite of well-known edge cases, but since we modify Zarith in a rather unorthodox way, it might miss some issues. Ideally, it would be possible to generate random input data and compare the output of the patched version with the original. However, the way we modify and distribute Zarith makes it impossible for these two versions to coexist. As a result, in my development branch, I created the alternative build targets "gobzarith.cmxa" and

"libgobzarith.a". These will rename modules and primitive functions to avoid symbol clashes, making it possible to test the different implementations of Zarith with QCheck.

A. Benchmarks

The benchmarks were performed on a server with an Intel Xeon E3-1270 v3 CPU and 32 GBs of RAM running Ubuntu 18.04. All dependencies of Goblint (with the exception of Zarith) were locked to the versions specified in the file "goblint.opam.locked" from commit "85536ab5" of the "goblint/analyzer" repository. Since I forked Zarith off the release 1.12, all builds of Goblint (including the "unmodified" ones) use this release for reasons of fairness.

Two separate benchmarks were performed. The goal of the first benchmark was to determine the impact each possible patch to Zarith had on performance. For this benchmark, I used the single-threaded program group. (see "index/single-thread.txt" in "goblint/bench") The second benchmark was done to ensure that the "winner" of the previous benchmark doesn't misbehave in other cases. For this reason, I only compared two different builds of Goblint (i.e., with unmodified Zarith & with the 53-bit version of Zarith). Since I used the tracing tests (see "index/traces.txt" in "goblint/bench") for this benchmark, Goblint was updated to "3690e97b," but its dependencies were left untouched.

```
1440 CAMLprim value ml_z_mul_overflows(value vx, value vy)
1441 {
1442 #if HAS_BUILTIN(__builtin_mul_overflow) || __GNUC__ >= 5
1443   intnat z;
1444   return Val_bool(__builtin_mul_overflow(vx - 1, vy >> 1, &z));
1445 #elif defined(__GNUC__) && defined(__x86_64__)
1446   intnat z;
1447   unsigned char o;
1448   asm("imulq %1, %3; seto %0"
1449       : "=q" (o), "=r" (z)
1450       : "1" (vx - 1), "r" (vy >> 1)
1451       : "cc");
1452   return Val_int(o);
1453 #elif defined(_MSC_VER) && defined(_M_X64)
1454   intnat hi, lo;
1455   lo = _mul128(vx - 1, vy >> 1, &hi);
1456   return Val_bool(hi != lo >> 63);
1457 #else
1458   /* Portable C code */
1459   intnat x = Long_val(vx);
1460   intnat y = Long_val(vy);
1461   /* Quick approximate check for small values of x and y.
1462      Also catches the cases x = 0, x = 1, y = 0, y = 1. */
1463   if (Z_FITS_HINT(x)) {
1464     if (Z_FITS_HINT(y)) return Val_false;
1465     if ((uintnat) x <= 1) return Val_false;
1466   }
1467   if ((uintnat) y <= 1) return Val_false;
1468 #if 1
1469   /* Give up at this point; we'll go through the general case in ml_z_mul */
1470   return Val_true;
```

Listing 14: Definition of "ml_z_mul_overflows"