From Bigints to Native Code

with ACL2 and

(well, ostensibly, anyway)

Jared Davis
ACL2 Seminar, 2016-03-29

github.com/jaredcdavis/acl2/
nativearith branch
Initial motivation

Hardware Design → VL, SV → (& a (* b c)) → Proofs, Symbolic Reasoning
Initial motivation

Hardware Design

VL, SV

(& a (* b c))

Proofs, Symbolic Reasoning

Linter

Viewer

Refactoring
Initial motivation

Hardware Design

VL, SV

Simulator? (Not really needed, but...)

(& a (* b c))

Linter

Viewer

Proofs, Symbolic Reasoning

Refactoring
Concrete evaluation

\[ \text{Svex-eval-list}((\& a (* b c)), \text{inputs}) \rightarrow \text{outputs} \]
Concrete evaluation

Horribly-Slow-
Svex-eval-list((& a (* b c)), inputs) \rightarrow outputs
Concrete evaluation

Horribly-Slow-
Svex-eval-list((& a (* b c)), inputs) → outputs

Fast alist
(hashing, allocation/gc)

List
(allocation/gc)
Concrete evaluation

Memoization (hashing, allocation/gc)

Cons tree (pointer chasing)

Horribly-Slow-
Svex-eval-list(
(& a (* b c))
, inputs) → outputs

Fast alist
(hashing, allocation/gc)

List (allocation/gc)
Concrete evaluation

Memoization (hashing, allocation/gc)
Cons tree (pointer chasing)

Horribly Slow Svex-eval-list( (& a (* b c)) , inputs) → outputs

Fn cases (interpretation)
Bignum arith (overflow checking, allocation/gc)

Fast alist (hashing, allocation/gc)
List (allocation/gc)
Sensible approach

Svex-eval-list((& a (* b c)), inputs) → outputs

Fast-eval-list(Stobj Array, Stobj Array) → Stobj Array

Lots better...

- Pointer chasing → Array accesses
- Hashing → Array accesses
- Memoization → Bit marking

But...

- fnccall interpretation
- bignum arithmetic
- lisp
This talk

- Native expression language
- LLVM connection
- Bigint representation
- Verified bigint operations
- Bounding bigint operations
- Big expression compiler
SV Expressions

Bigint Expressions

“Native” Expressions

LLVM Assembly

ACL2

in array

CFFI

libxx.so

out array
Small expressions

Smallexpr ::= Const val
    | Var name
    | Call fn args

\[\begin{align*}
\text{Const} \; \text{val} \mid_{\text{env}} &= \text{val} \\
\text{Var} \; \text{name} \mid_{\text{env}} &= \text{env}[\text{name}] \\
\text{Call} \; \text{fn} \; \text{args} \mid_{\text{env}} &= \text{fn}(\text{arg}_1 \mid_{\text{env}}, ..., \text{arg}_N \mid_{\text{env}}) \\
\end{align*}\]

FTY, Hons, memoization
Fixing conventions

\textbf{Const} \, \texttt{val}_\text{\textless env} = \texttt{val} \quad \text{Bad values get fixed}

\textbf{Var} \, \texttt{name}_\text{\textless env} = \texttt{env}[\texttt{name}] \quad \text{Missing vars get 0'd}

\textbf{Call} \, \texttt{fn \, args}_\text{\textless env} = \texttt{fn}(\texttt{arg}_1\,\text\textless env, ..., \texttt{arg}_N\,\text\textless env) \quad \text{Extra args are ignored}

\text{Missing args get 0'd}

\text{Unknown functions return 0}
Values and operations

64-bit integers everywhere
ACL2 representation is signed (i64-p)
Still can do unsigned ops (i64slt vs i64ult)
Unityped expressions

Total, wraparound operations
\[ A/0 = 0 \quad -(-2^{63}) = -2^{63} \quad -2^{63} / -1 = -2^{63} \]

Comparisons return 0 or 1
Separate operation for addition carryout
Operations produce 64-bit results

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ a b)</td>
<td>Addition</td>
</tr>
<tr>
<td>(+ cin a b)</td>
<td>Addition</td>
</tr>
<tr>
<td>(- a)</td>
<td>Subtraction</td>
</tr>
<tr>
<td>(- a b)</td>
<td>Subtraction</td>
</tr>
<tr>
<td>(- a 1)</td>
<td>Subtraction</td>
</tr>
<tr>
<td>(abs a)</td>
<td>Absolute value</td>
</tr>
<tr>
<td>(logcdr a)</td>
<td>Logarithm of the difference</td>
</tr>
<tr>
<td>(loghead n a)</td>
<td>Logarithm of the head</td>
</tr>
<tr>
<td>(logtail n a)</td>
<td>Logarithm of the tail</td>
</tr>
<tr>
<td>(lognot a)</td>
<td>Logical complement</td>
</tr>
<tr>
<td>(+ 1 (lognot a))</td>
<td>Addition of one and logical complement</td>
</tr>
<tr>
<td>(ash x a)</td>
<td>Arithmetic shift</td>
</tr>
<tr>
<td>(* a b)</td>
<td>Multiplication</td>
</tr>
<tr>
<td>(truncate a b)</td>
<td>Truncate</td>
</tr>
<tr>
<td>(rem a b)</td>
<td>Remainder</td>
</tr>
<tr>
<td>(floor a b)</td>
<td>Floor</td>
</tr>
<tr>
<td>(mod a b)</td>
<td>Modulo</td>
</tr>
</tbody>
</table>
Aside - Collecting variables

(defines smallexpr-vars

(define smallexpr-vars ((x smallexpr-p))
  :returns (vars smallvarlist-p)
  (smallexpr-case x
    :const nil
    :var  (list x.var)
    :call (smallexprlist-vars x.args)))

(define smallexprlist-vars ((x smallexprlist-p))
  :returns (vars smallvarlist-p)
  (if (atom x)
    nil
    (append (smallexpr-vars (car x))
      (smallexprlist-vars (cdr x))))))

Logically simple, but inefficient
Approaches

ACL2 memoization, ordered sets (aig, 4v-sexpr, svex)
  Easy reasoning
  Big variable lists all over

Spare bitsets (4v-nsexprs), essentially the same

Explicit seen table (aig, 4v-sexpr, svex)
(define smallexpr-vars-memo ((x smallexpr-p) seen ans)
  :returns (mv new-seen new-ans)
  (b* ((kind (smallexpr-kind x))
    ((when (eq kind :const)) ;; trivial, don't mark
      (mv seen ans))
    ((when (hons-get x seen))
      (mv seen ans))
    (seen (hons-acons x t seen))
    ((when (eq kind :var))
      (mv seen (cons (smallexpr-var->var x) ans))))
  (smallexprlist-vars-memo (smallexpr-call->args x)
    seen ans)))

(define smallexprlist-vars-memo ((x smallexprlist-p) seen ans)
  :returns (mv new-seen new-ans)
  (b* (((when (atom x))
      (mv seen ans))
    (mv seen ans)
    (mv seen ans)
    (smallexpr-vars-memo (car x) seen ans)))
  (smallexprlist-vars-memo (cdr x) seen ans)))
(define smallexpr-vars-memo ((x smallexpr-p) seen ans)
  :returns (mv new-seen new-ans)
  (b* ((kind (smallexpr-kind x))
        ((when (eq kind :const)) ;; trivial, don't mark
         (mv seen ans))
        ((when (hons-get x seen))
         (mv seen ans))
        (seen (hons-acons x t seen)))
        ((when (eq kind :var))
         (mv seen (cons (smallexpr-var->var x) ans))))
    (smallexprlist-vars-memo (smallexpr-call->args x)
     seen ans)))

(define smallexprlist-vars-memo ((x smallexprlist-p) seen ans)
  :returns (mv new-seen new-ans)
  (b* (((when (atom x))
     (mv seen ans))
     (mv seen ans)
    (smallexpr-vars-memo (car x) seen ans)))
    (smallexprlist-vars-memo (cdr x) seen ans)))
Preorder marking invariant

Marking x seen before we recur breaks the obvious invariant,

For all nodes N we have SEEN
(smallexpr-vars N) are all in ANS

Crux: we can mark x as seen before we visit its children because x is “bigger” than its (transitive) children.

The proof has been done over and over again (4v, sv, ...). It is horrible and tedious.
Generic proof

(nc-node-children node) → children
How to get the children from a node

(nc-node-elems node) → elems
Get the elements to collect from a single node

(nc-node-count node) → count
(nc-nodelist-count node) → count
Measures to ensure termination

(nct-node-trivial node) → bool
SV Expressions → Bigint Expressions → “Native” Expressions → LLVM Assembly

ACL2

in array

out array

CFFI

libxx.so
LLVM operations

```cpp
define i64 @narith_i64bitand (i64 %a, i64 %b) {
    %ans = and i64 %a, %b
    ret i64 %ans
}

define i64 @narith_i64eql (i64 %a, i64 %b) {
    %ans = icmp eq i64 %a, %b
    %ext = zext i1 %ans to i64
    ret i64 %ext
}
```
define i64 @narith_i64sdiv (i64 %a, i64 %b) {
    %b.zero = icmp eq i64 %b, 0
    br i1 %b.zero, label %case.zero, label %case.nonzero

  case.nonnzer:
    %a.intmin = icmp eq i64 %a, -9223372036854775808
    %b.minus1 = icmp eq i64 %b, -1
    %overflow = and i1 %a.intmin, %b.minus1
    br i1 %overflow, label %case.overflow, label %case.usual

  case.zero:
    ret i64 0

  case.overflow:
    ret i64 %a

  case.usual:
    %ans = sdiv i64 %a, %b
    ret i64 %ans
}
Validating LLVM operations

ops.lisp
(define narith_i64sdiv ((a i64-p) (b i64-p))
(progn$
 (raise "LLVM definition not installed?")
(i64sdiv a b)))

ops-raw.lsp
(cffi:defcfun "narith_i64sdiv"
  Returns :int64
  Takes (a :int64) (b :int64))

ops.ll
(define narith_i64sdiv ((a i64-p) (b i64-p))
(progn$
 (raise "LLVM definition not installed?")
(i64sdiv a b)))

libnarith_ops.so

CFFI

include-raw
Validating LLVM operations

\[
\text{narith}_\text{i64sdiv}
\]

\[
(\text{cond ((eql b 0)
  0)
  ((and (eql b -1)
    (eql a (- (expt 2 63))))
    a)
  (t
    (the (signed-byte 64)
      (truncate a b)))))
\]

\text{opstest.lisp}

crafted tests
with boundary cases
+ 100k random tests

\text{libnarith_ops.so}

\text{cffi:defcfun ...}
Compiling small expressions

(smallexprlist-eval exprs env) = outs

void my_circuit (i64* ins, i64* outs);
[ a, (* b c), (& a (+ (* b c) 5)) ]
Assign input indices (%i)

\[
[ a, (* b c), (& a (+ (* b c) 5)) ]
\]
Assign input indices (%i)
Assign output indices (%o)

\[
[\ a,\ (*\ b\ c),\ (&\ a\ (+\ (*\ b\ c\ 5)))\ ]
\]
Assign input indices (%i)
Assign output indices (%o)
Assign node numbers (%n)

\[
[ a, (* b c), (& a (+ (* b c) 5)) ]
\]
Variables
\%i0 = getelementptr i64* \%in, i32 0
\%n0 = load i64* \%i0

Constants
\%n4 = add i64 0, 5

Functions
\%n3 = call i64 @narith_times(i64 \%n1, i64 \%n2)
Outputs

%i0 = getelementptr i64* %in, i32 0
%n0 = load i64* %i0
%o0 = getelementptr i64* %out, i32 0
store i64 %n0, i64* %o0

%n3 = call i64 @narith_times(i64 %n1, i64 %n2)
%o1 = getelementptr i64* %out, i32 0
store i64 %n3, i64* %o1
Compiler quality, or lack thereof...

Calls of @narith_foo get nicely inlined

Useless ITE branches still get computed

%n approach uses LLVM's register allocator, but seems to cause a lot of spilling

I can't get it to smartly reorder instructions to avoid register spilling
Gluing it together

SV Expressions

Bigint Expressions

“Native” Expressions

LLVM Assembly

Array in

Array out

CFFI

libxx.so
(smallexprs-to-llvm-top
"foo"
(list '(i64eql a b)
     '(i64plus a (i64bitand b c))))

(:llvmasm
(FNNAME . "foo")
(CODE . "... ops definitions ...
define void @foo (i64* noalias nocapture align 8 %in,
     i64* noalias nocapture align 8 %out)
{
   %i0 = getelementptr i64* %in, i32 0
   %n1 = load i64* %i0, align 8
   %i1 = getelementptr i64* %in, i32 1
   %n2 = load i64* %i1, align 8
   %n0 = call i64 @narith_i64eql(i64 %n1,i64 %n2)
   %o0 = getelementptr i64* %out, i32 0
   store i64 %n0, i64* %o0, align 8, !nontemporal !{i32 1}
   %i2 = getelementptr i64* %in, i32 2
   %n5 = load i64* %i2, align 8
   %n4 = call i64 @narith_i64bitand(i64 %n2,i64 %n5)
   %n3 = call i64 @narith_i64plus(i64 %n1,i64 %n4)
   %o1 = getelementptr i64* %out, i32 1
   store i64 %n3, i64* %o1, align 8, !nontemporal !{i32 1}
   ret void
}"
) ...)
Top level wrapper (wip)

Create assembly to compile, write to tmpdir/foo.ll

Tshell runs: (some better sequence?)
  opt -O3 foo.ll foo.ll.opt
  llc -o foo.ll.s foo.ll.opt
  as foo.ll.s -o foo.o
  clang -shared foo.o -o libfoo.so (really???)

Then load library

Still working out top-level wrapper:
  Loading the library works fine
  Need to develop stobj interface
  (Q: can we avoid any copying penalty?)
SV Expressions

Bigint Expressions

“Native” Expressions

LLVM Assembly

ACL2

libxx.so

in array

out array

CFFI
Big integer representation

Uh... integerp?
 Explicit representation

(list a b c ... z)

Least Significant

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>...</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>64</td>
<td>64</td>
<td></td>
<td>64</td>
</tr>
</tbody>
</table>

Most Significant

Bit 63 Sign Extended

Representation: signed 64-bit integers, **but** only the most significant is treated as signed
Examples

'(5 7 9) encodes 5 + 7 \cdot 2^{64} + 9 \cdot 2^{128}

'(5 7 -1) encodes 5 + 7 \cdot 2^{64} + -1 \cdot 2^{128}

'(-1 7) encodes 2^{64} - 1 + 7 \cdot 2^{64}
Not a canonical representation

'(5 7) encodes $5 + 7 \cdot 2^{64}$

'(5 7 0) encodes $5 + 7 \cdot 2^{64}$

'(5 7 0 0) encodes $5 + 7 \cdot 2^{64}$

Useful: N blocks can represent any $64 \cdot N$ bit int
(define bigint-p (x)  
   :returns (ans booleanp)  
   (and (consp x)  
        (i64list-p x)))

(define bigint-fix (x) ...)  
(define bigint-equiv (x y) ...)  
(define bigint-singleton (a) ...)  
(define bigint-cons (a x) ...)
(define bigint->endp ((x bigint-p))
  :returns (endp booleanp)
  (let ((x (bigint-fix x)))
    (atom (cdr x))))

(define bigint->first ((x bigint-p))
  :returns (first i64-p)
  (let ((x (bigint-fix x)))
    (i64-fix (car x))))

(define bigint->rest ((x bigint-p))
  :returns (rest bigint-p)
  (let ((x (bigint-fix x)))
    (if (consp (cdr x))
      (cdr x)
      (let ((first (bigint->first x)))
        (if (< first 0)
          (bigint-minus1)
          (bigint-0)))))))
(define bigint->val ((x bigint-p))
  :returns (val integerp)
  (if (bigint->endp x)
      (bigint->first x)
      (logapp 64
        (bigint->first x)
        (bigint->val (bigint->rest x))))
B* Integration

(make-event
  (std::da-make-binder 'bigint '(first rest endp)))

(defrule patbind-bigint-example
  (b* ((bigint x))
    (and (equal x.first (bigint->first x))
      (equal x.rest (bigint->rest x))
      (equal x.endp (bigint->endp x)))))
Big integer operations

The cheater way

(define bigint-lognot ((a bigint-p))
 :returns (ans bigint-p)
 (make-bigint (lognot (bigint->val a))))
Big integer operations

The cheater way

```
(define bigint-lognot ((a bigint-p))
  :returns (ans bigint-p)
  (make-bigint (lognot (bigint->val a))))
```
Without cheating

\[
\text{(define bigint-lognot \((a \text{ bigint-p})\)) :returns \((\text{ans bigint-p})\)}
\]
\[
(\text{b* \(((\text{bigint a}))\)}
\]
\[
(\text{first (i64bitnot a.first)})
\]
\[
((\text{when a.endp})
\]
\[
(\text{bigint-singleton first)})
\]
\[
(\text{bigint-cons first (bigint-lognot a.rest))})
\]

Small operations

Explicit construction of bigint blocks

\[
(\text{defrule bigint-lognot-correct})
\]
\[
(\text{equal \((\text{bigint->val (bigint-lognot a)})\)}
\]
\[
(\text{lognot (bigint->val a)})))}
Easy operations

Bitwise operations
  Recur down args and combine

Equal/unequal
  Recur down args until they end/disagree

Less-than/etc
  Just check more significant blocks first
Building blocks for plus

(define i64plus ((a i64-p) (b i64-p))
  (b* ((a (logext 64 a))
       (b (logext 64 b)))
       (logext 64 (+ a b))))

(define i64upluscarry ((a i64-p) (b i64-p))
  (b* ((a (loghead 64 a))
       (b (loghead 64 b)))
       (bool->bit
        (not (unsigned-byte-p 64 (+ a b))))))

Each has a nice LLVM definition
Key lemma for plus (general form)

(defrule split-plus
  (implies
    (bitp cin)
    (equal (logapp n
      (+ cin
        (loghead n a)
        (loghead n b))
      (+ (plus-ucarryout-n n cin a b)
        (logtail n a)
        (logtail n b)))
    (+ cin
      (ifix a)
      (ifix b))))

Carryout from
Cin + a.first + b.first

Low n bits +
High n bits =
Full sum (+ cin
  (ifix a)
  (ifix b)))
(defrule split-plus
  (implies
    (bitp cin)
    (equal (logapp n
      (+ cin
        (loghead n a)
        (loghead n b))
      (+ (plus-ucarryout-n n cin a b)
        (logtail n a)
        (logtail n b)))
    (+ cin
      (ifix a)
      (ifix b))))
(logapp n
  (+ cin (loghead n a) (loghead n b))
  (+ (plus-ucarryout-n n cin a b)
      (logtail n a)
      (logtail n b)))
(logapp n
  (+ cin (loghead n a) (loghead n b))
  (+ (plus-ucarryout-n n cin a b) (logtail n a) (logtail n b)))

(define bigint-plus-sum0 ((cin    bitp) (afirst i64-p) (bfirst i64-p))
  (b* ((cin     (lbfix cin))
       (afirst  (i64-fix afirst))
       (bfirst  (i64-fix bfirst))
       (cin+a   (i64plus cin   afirst))
       (cin+a+b (i64plus cin+a bfirst)))
  cin+a+b))

(defrule bigint-plus-sum0-correct
  (equal (bigint-plus-sum0 cin afirst bfirst) (logext 64 (+ (bfix cin)
               (i64-fix afirst) (i64-fix bfirst))))))
(logapp n
 (+ cin (loghead n a) (loghead n b))
 (+ (plus-ucarryout-n n cin a b)
 (logtail n a)
 (logtail n b)))

(define bigint-plus-cout0 ((cin     bitp)
 (afirst  i64-p)
 (bfirst  i64-p))
 (b* ((cin     (lbfix cin))
 (afirst  (i64-fix afirst))
 (bfirst  (i64-fix bfirst))
 (cout    (i64uplususcarry ??? ???)))
cout)
(logapp n (+ cin (loghead n a) (loghead n b)) (+ (plus-ucarryout-n n cin a b) (logtail n a) (logtail n b)))

(define bigint-plus-cout0 ((cin bitp) (afirst i64-p) (bfirst i64-p))
  (b* ((cin (lbfix cin))
       (afirst (i64-fix afirst))
       (bfirst (i64-fix bfirst))
       (cin+a (i64plus cin afirst))
       (cout (i64uplusucarry cin+a bfirst)))
  cout))
Wrong exactly when $A = -1$ and $CIN = 1$

We get 0 instead of 1
Example

\[
\begin{array}{c}
\text{Cin} \\
\text{A} \quad 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \\
\text{B} \quad 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0
\end{array}
\]
Cin

A

B

0 1 0 1 0 1 0 1 0

1

0
<table>
<thead>
<tr>
<th>Cin</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 1</td>
<td>1 0</td>
</tr>
</tbody>
</table>

The result is 1 0.
\[(\logapp n \n
\begin{align*}
\text{(plus-ucarryout-n n cin a b)}
\end{align*}
\)\)
(define bigint-plus-aux ((cin bitp)
  (a bigint-p)
  (b bigint-p))

  :returns (ans bigint-p)
  (b* ((cin (lbfix cin))
        ((bigint a))
        ((bigint b))
        (sum0 (bigint-plus-sum0 cin a.first b.first))
        (cout (bigint-plus-cout0 cin a.first b.first))
        ((when (and a.endp b.endp))
         (b* ((asign  (bigint->first a.rest))
              (bsign  (bigint->first b.rest))
              (final  (i64plus cout (i64plus asign bsign))))))
        (bigint-clean (bigint-cons sum0
                         (bigint-singleton final)))))

(bigtint-cons sum0
  (bigint-plus-aux cout a.rest b.rest)))

(defrule bigint-plus-aux-correct
  (equal (bigint->val (bigint-plus-aux cin a b))
         (+ (bfix cin) (bigint->val a) (bigint->val b))))
Top level plus

(define bigint-plus ((a bigint-p) (b bigint-p))
  :returns (ans bigint-p)
  (bigint-plus-aux 0 a b))

(defrule bigint-plus-correct
  (equal (bigint->val (bigint-plus a b))
         (+ (bigint->val a) (bigint->val b))))
Aside: proving split-plus

(define recursive-plus ((cin bitp)
  (a integerp)
  (b integerp))

(b* (((when (and (or (zip a) (eql a -1))
      (or (zip b) (eql b -1))))
     (recursive-plus-base-case cin a b))
   (a0 (logcar a))
   (b0 (logcar b))
   (sum (b-xor cin (b-xor a0 b0)))
   (cout (b-ior (b-and a0 b0)
                (b-and cin (b-ior a0 b0))))
   (logcons sum
            (recursive-plus cout
                           (logcdr a)
                           (logcdr b))))
(define plus-ucarryout-n ((n  natp)  
  (cin  bitp)  
  (a   integerp)  
  (b   integerp))  

:returns (cout  bitp)  
(b* (((when (zp n))  
  (bfix cin))  
  (a0   (logcar a))  
  (b0   (logcar b))  
  (cout (b-ior (b-and a0 b0)  
               (b-and cin (b-ior a0 b0))))))  

(plus-ucarryout-n  
  (- n 1) cout (logcdr a) (logcdr b)))
(define recursive-plus-base-case ((cin bitp)
  (a  integerp)
  (b  integerp))

(let ((a0 (logcar a))
  (b0 (logcar b)))
 (logcons (b-xor cin (b-xor a0 b0))
  (logext 1 (b-ior (b-and a0 b0)
    (b-and (b-xor a0 b0)
      (b-not cin))))))

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>
On beyond plus

(define bigint-minus ((a bigint-p) (b bigint-p))
  (bigint-plus-aux 1 a (bigint-lognot b)))

(defrule bigint-minus-correct
  ;; Via bitops::minus-to-lognot
  (equal (bigint->val (bigint-minus a b))
    (- (bigint->val a)
      (- (bigint->val a)
        (bigint->val b))))
Loghead/logext

\[(\text{bigint-loghead } n \ a) \rightarrow a.\text{first} :: (\text{bigint-loghead } (- n 64) \ a.\text{rest})\]

Need subtraction
Special error for negative A, huge N
Special early-out for positive A, huge N

Logext is similar

Other operations (times, divide, shift, etc.) are all TODO
SV Expressions

Bigint Expressions

“Native” Expressions

LLVM Assembly

in array

CFFI

libxx.so

out array

ACL2
Bigexpr compiler (wip)

Compile(bigexprs, varsizes) → smallexpr lists * varmap

Size bounds for input variables

Each bigexpr → List of smallexpr

How to translate environments
Bounding expressions

Bound bigexprs, starting from bounds for the variables (given by the user)

Bigbound structures: size, min/max (optional)

Bounders for all current bigint functions
- Certify very fast (compare to tau!)
- Avoid unreasonably large bounds
- Sometimes not very good
Aside - Pretty goals

(IMPLIES (AND (SIGNED-BYTE-P (BIGBOUND->SIZE BOUND1) 
   (BIGINT->VAL (BIEGEVAL ARG1 ENV))) 
   (NOT (BIGBOUND->MIN BOUND1)) 
   (<= (BIGINT->VAL (BIEGEVAL ARG1 ENV)) 
       (BIGBOUND->MAX BOUND1))) 
(SIGNED-BYTE-P (BIGBOUND->SIZE BOUND2) 
   (BIGINT->VAL (BIEGEVAL ARG2 ENV))) 
(<= (BIGBOUND->MIN BOUND2) 
   (BIGINT->VAL (BIEGEVAL ARG2 ENV))) 
(NOT (BIGBOUND->MAX BOUND2)) 
(BIGBOUND->MAX BOUND1) 
(< 0 (BIGBOUND->MAX BOUND1)) 
(BIGBOUND->MIN BOUND2) 
(<= 0 (BIGBOUND->MIN BOUND2)) 
(<= (+ 1 (BIGBOUND->MAX BOUND1)) 
   (BIGBOUND->SIZE BOUND2))) 
(<= (LOGHEAD (BIGINT->VAL (BIEGEVAL ARG1 ENV)) 
   (BIGINT->VAL (BIEGEVAL ARG2 ENV))) 
(+ -1 (ASH 1 (BIGBOUND->SIZE BOUND1)))))
(B* (((BIGBOUND BOUND1))
  ((BIGBOUND BOUND2)))
(IMPLIES (AND BOUND1.MAX BOUND2.MIN (NOT BOUND1.MIN)
  (NOT BOUND2.MAX)
  (< 0 BOUND1.MAX)
  (<= 0 BOUND2.MIN)
  (<= (+ 1 BOUND1.MAX) BOUND2.SIZE)
  (<= BOUND2.MIN
      (BIGINT->VAL (BIGEVAL ARG2 ENV)))
  (<= (BIGINT->VAL (BIGEVAL ARG1 ENV))
      BOUND1.MAX)
  (SIGNED-BYTE-P BOUND1.SIZE
      (BIGINT->VAL (BIGEVAL ARG1 ENV)))
  (SIGNED-BYTE-P BOUND2.SIZE
      (BIGINT->VAL (BIGEVAL ARG2 ENV))))
  (<= (LOGHEAD (BIGINT->VAL (BIGEVAL ARG1 ENV))
      (BIGINT->VAL (BIGEVAL ARG2 ENV)))
  (+ -1 (ASH 1 BOUND1.SIZE))))

(include-book "tools/prettygoals/top" :dir :system)
Other resources

XDOC manual

github.com/jaredcdavis/acl2
(nativearith branch)
Conclusions

Notes on arithmetic reasoning

Introducing new languages still tricky

Thoughts about other applications
Thanks!