Hardware to Software Symbolic Simulation with FGL

Sol Swords, Arm

What is FGL?

- A bitblasting symbolic simulation framework
  - Hardware datapath spec function → Boolean formulas for equiv checking
- A rewriter modeled after ACL2's
  - Extended with useful features for symbolic simulation

What is this talk about?

- Development of FGL around hardware verification
  - History at Centaur, Intel
  - Use for architecture pseudocode validation at Arm
- Fun things to do with your custom rewriter

Outline

- FGL history
  - G System, GL
  - Design ambitions of FGL
- Hardware to Software transition
  - Current project at Arm
  - Why FGL?
- Problem Solving in Software FV
  - Using type information in symbolic simulation
  - Debugging counterexamples of abstractions
  - Symbolic simulation under no-error assumptions

History

G to GL to FGL

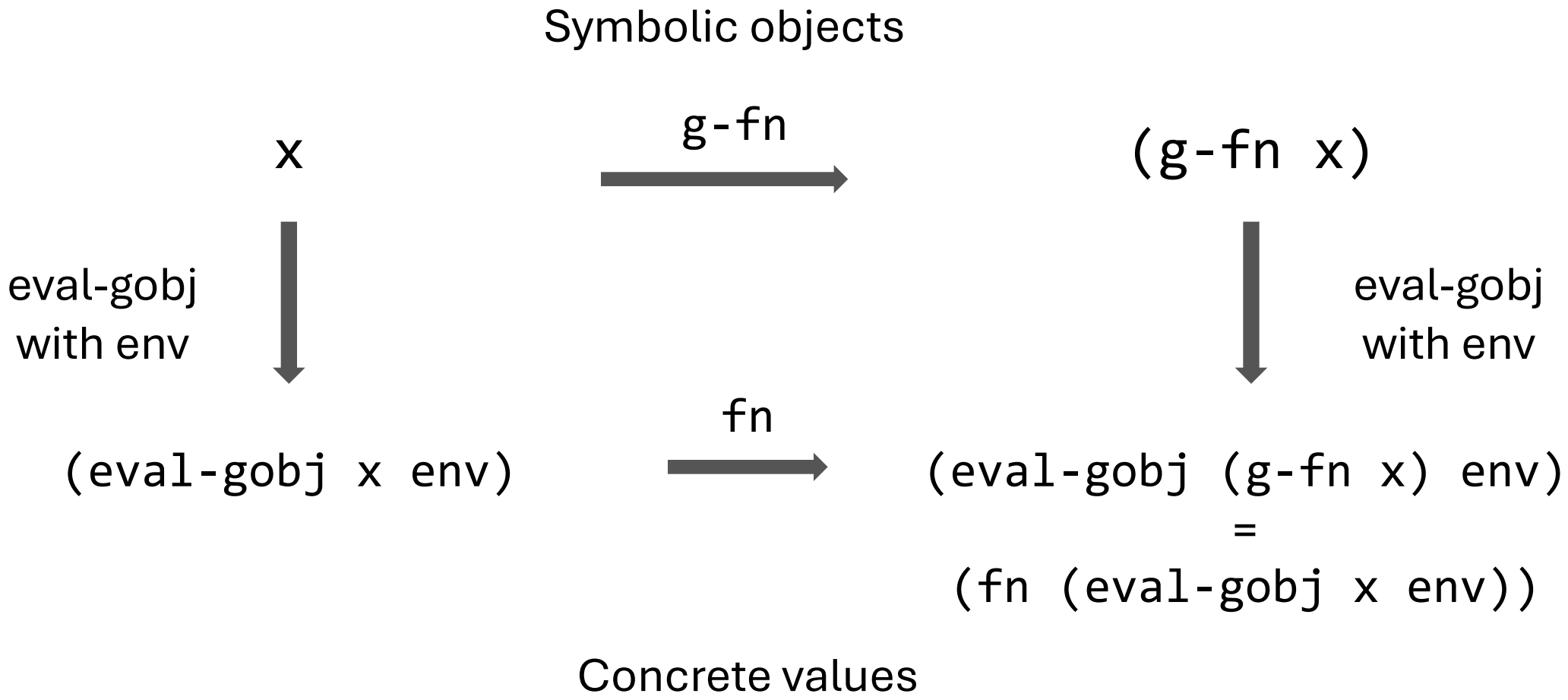
History: G to GL

- The G System: Boyer & Hunt -- developed at Centaur to build BDDs for floating point addition spec
  - Raw Lisp routine compiled ACL2 functions into BDD-based “symbolic counterparts”
- GL (“G in the Logic”)* -- out of raw Lisp
  - Verifying compilation of ACL2 functions into ACL2 BDD-based symbolic counterparts
  - Straightforward commutative diagram correctness property

    (eval-gobj (g-fn x) env) = (fn (eval-gobj x env))

* Names are the worst

GL correctness: both paths agree



GL Evolution

- Drawbacks of symbolic counterpart compiler
  - Theorem conclusion needed to be compiled as well
  - Automation fairly robust, but verification could still fail
- Added symbolic interpreter—made compiler nearly useless
  - Compiler still exists vestigially
  - “Primitive” functions still have hand-coded symbolic counterparts
  - Everything else is interpreted
  - Verified (slightly more complex commutative diagram)
  - Bonus: can use a theorem other than a function’s definition to expand it
    - In fact, interpreter quickly turned into rewriter – just add relieve-hyps, unify

GL → FGL*: Design ambitions

GL features	FGL ambitions
Built-in primitives; reprogram and rebuild	User-driven rules, more rewriting capabilities allowing for less syntactic manipulation
Hash-cons-based BDDs and AIGs	Stobj-based Boolean function representation with AIGNET
Rewriter state on the Lisp stack	Explicit rewriter state data structure; easier inspection and debugging
Input symbolic objects and BDD order derived from shape specifiers	Inputs are just variables Boolean expressions introduce Boolean variables
	Rewrite-driven SAT Including incremental SAT
	Global AIG transformations At any time: DAG-aware rewriting and SAT sweeping across all Boolean expressions
Both fully verified ACL2 clause processors	

*FGL = “Future GL”: names are the worst.

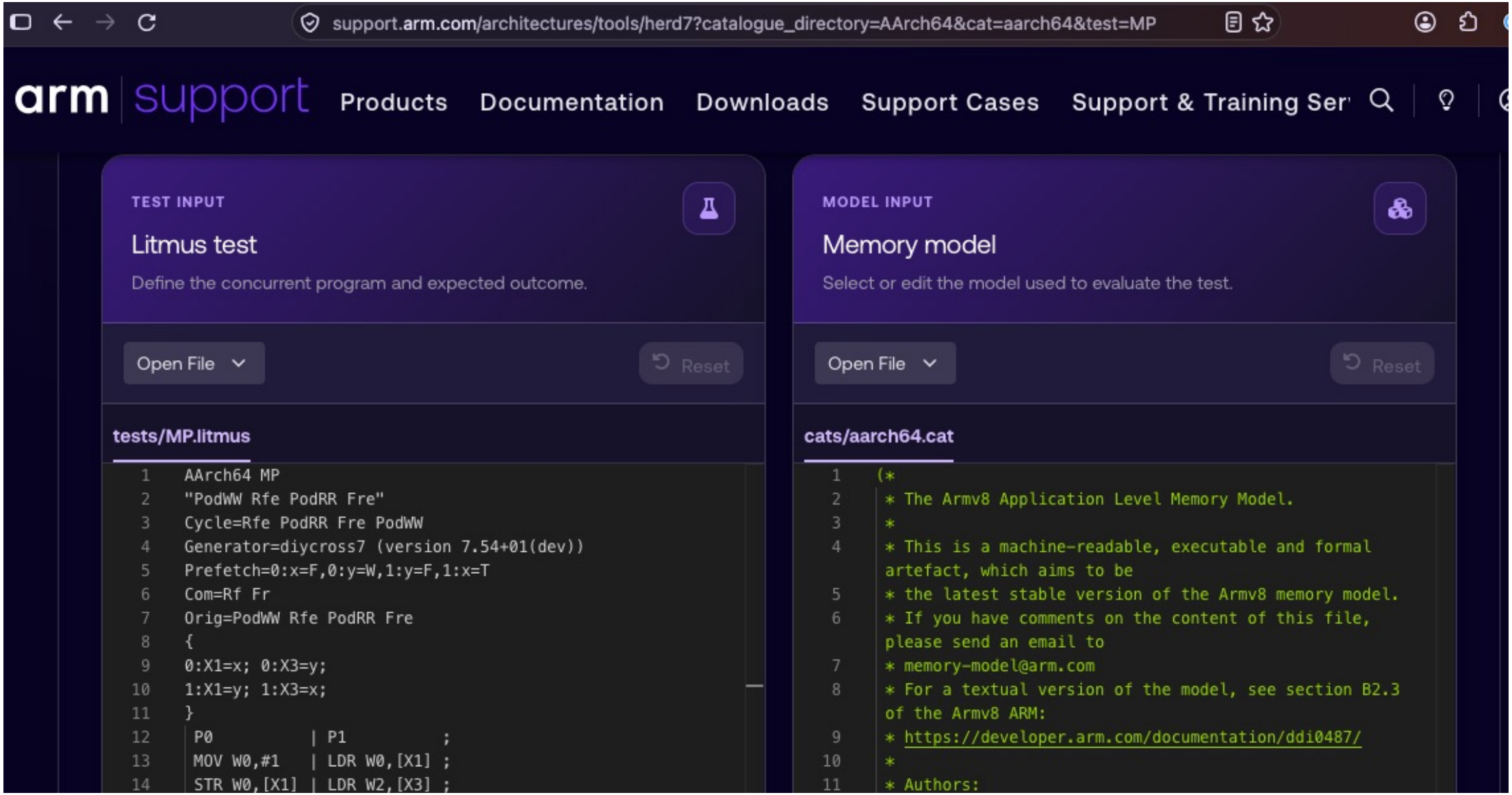
Hardware to Software

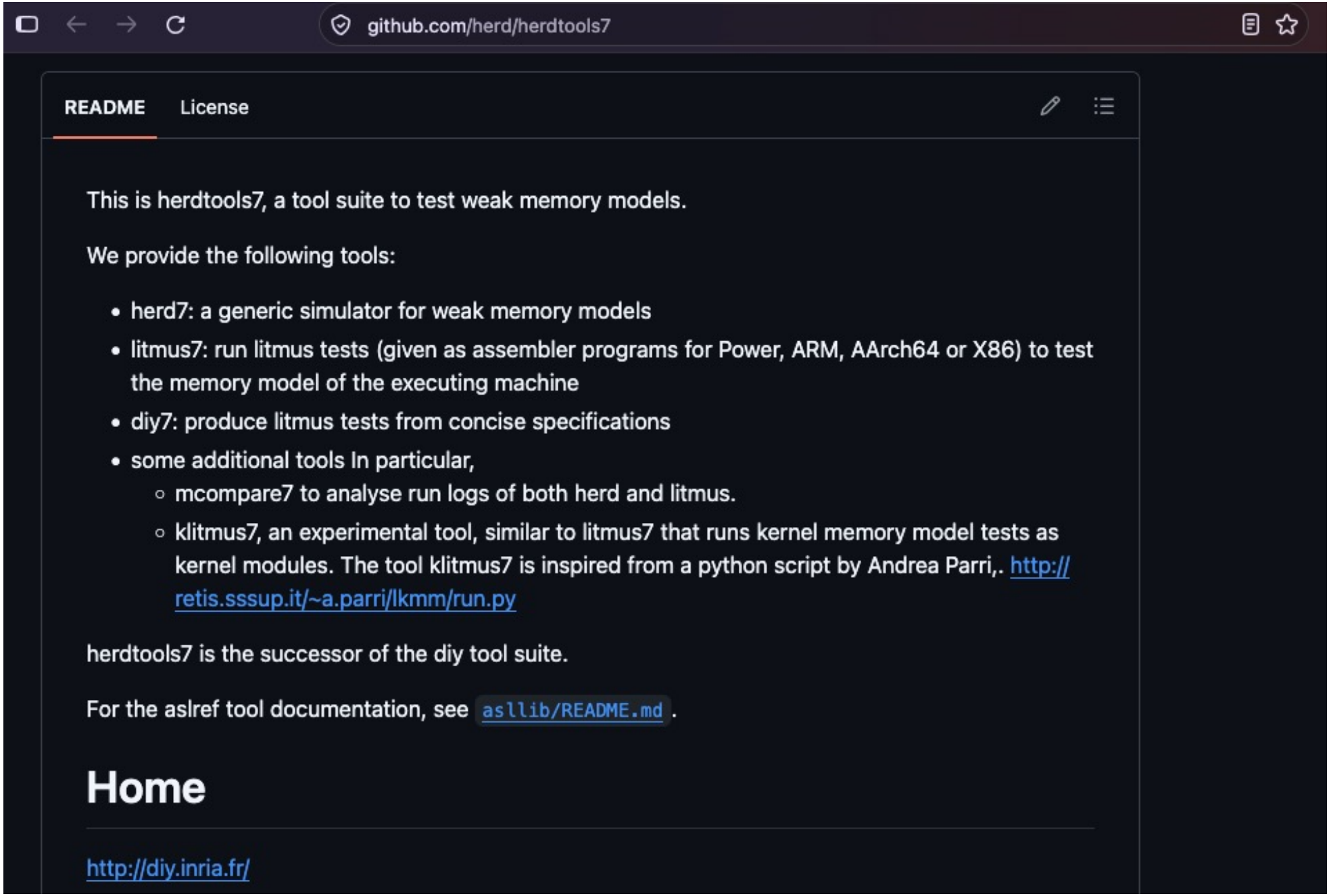
Hardware use

- Major proof engine for Centaur/Intel VL/SV toolsuite
  - Alongside Temel's VeSCMul rewriter for multipliers/adder trees
- Very successful at hardware datapath/arithmetic proofs
  - **"Robust, End-to-end Correctness Proofs of Industrial Divide and Square Root RTL Designs"** (Arith-2025)
- Typical usage was mainly to enable equivalence checking between hardware model and specs
  - Really just bitblasting
  - For majority of proofs this was all that was needed
  - For proofs requiring decomposition (divide, sqrt, FMA) relied on ACL2 rewriter for composition

Arm Architecture Formal Verification

- Cambridge-based group led by Jade Alglave
- Focus: clarity, accuracy, formality of Arm ISA specifications
- Major subprojects:
  - CAT specification of concurrent semantics
    - Mathematically describe what relations are allowed between concurrent operations
    - Tools to check whether outcomes of concurrent programs satisfy spec
  - Instruction specification pseudocode in ASL
    - Formalization of ASL language
    - Properties proved about instruction specifications
  - Accessibility of architecture info
    - “The Architecture Speaks” chatbot


support.arm.com/architectures/tools/herd7?catalogue_directory=AArch64&cat=aarch64&test=MP
arm support
Products
Documentation
Downloads
Support Cases
Support & Training Ser
TEST INPUT
Litmus test
Define the concurrent program and expected outcome.
Open File
Reset
tests/MP.litmus
1 AArch64 MP
2 "PodWW Rfe PodRR Fre"
3 Cycle=Rfe PodRR Fre PodWW
4 Generator=diycross7 (version 7.54+01(dev))
5 Prefetch=0:x=F,0:y=W,1:y=F,1:x=T
6 Com=Rf Fr
7 Orig=PodWW Rfe PodRR Fre
8 {
9 0:X1=x; 0:X3=y;
10 1:X1=y; 1:X3=x;
11 }
12 P0 | P1 ;
13 MOV W0,#1 | LDR W0,[X1] ;
14 STR W0,[X1] | LDR W2,[X3] ;
MODEL INPUT
Memory model
Select or edit the model used to evaluate the test.
Open File
Reset
cats/aarch64.cat
1 (*
2 * The Armv8 Application Level Memory Model.
3 *
4 * This is a machine-readable, executable and formal artefact, which aims to be
5 * the latest stable version of the Armv8 memory model.
6 * If you have comments on the content of this file, please send an email to
7 * memory-model@arm.com
8 * For a textual version of the model, see section B2.3 of the Armv8 ARM:
9 * https://developer.arm.com/documentation/ddi0487/
10 *
11 * Authors:


github.com/herd/herdtools7
README
License
This is herdtools7, a tool suite to test weak memory models.
We provide the following tools:
herd7: a generic simulator for weak memory models
litmus7: run litmus tests (given as assembler programs for Power, ARM, AArch64 or X86) to test the memory model of the executing machine
diy7: produce litmus tests from concise specifications
some additional tools In particular,
mcompare7 to analyse run logs of both herd and litmus.
klitmus7, an experimental tool, similar to litmus7 that runs kernel memory model tests as kernel modules. The tool klitmus7 is inspired from a python script by Andrea Parri,. http://retis.sssup.it/~a.parri/lkmm/run.py
herdtools7 is the successor of the diy tool suite.
For the aslref tool documentation, see asllib/README.md .
Home
http://diy.inria.fr/

Architecture Pseudocode Analysis

- Goal: prove properties of instruction spec ASL pseudocode
  - Certain assertions don’t fail
  - Instructions satisfy certain expectations e.g. do/do not set flags, do/do not check alignment
  - Equivalence/correspondence between refactors
- Instruction pseudocode is significant software
  - Big subroutines such as address translation, exception handling
  - Publicly released pseudocode is >200,000 LOC
- Language is formally specified but featureful, high-level
  - Statically typed with structured datatypes
  - Statements, expressions, loops, functions, …

Our Approach

- ASL interpreter written in ACL2
  - Based on formal semantics provided by ASLRef team
  - Runs on type-checked AST produced by ASLRef*
  - Proofs characterize results from interpreter applied to AST constant
- Main proof approach is symbolic simulation with FGL
  - Developed extensive FGL rewriting theory for ASL
  - Practically covers thousands of LOC per proof
- “Correctness”/sanity proofs for standard library functions
  - Using ACL2 rewriter/waterfall

*just this once, we didn’t write a parser in ACL2.

Why FGL vs ACL2 prover?

Rewrite everything once – no going back!

Advantages: speed and efficiency

- Doing one rewrite pass is fast and memory efficient
- Avoids clausification, which is death on big symbolic simulations
- Needed for scalability: thousands of function calls in some proofs

Requirements: rewriting discipline, theorem phrasing

- Rewriting depends on ordering of assumptions
  - If you need a rewrite to apply to a term, its assumptions need to be known (in path condition) before the term is reached – no second chance after clausification
  - Sometimes may need a manual case split
- Need to combine results from IF branches
  - Leaving rewritten term as an IF usually causes further rewriting to fail
  - Handle with branch merge rewrite rules

Using type information in symbolic simulation

Using type information in symbolic simulation

- Language uses a strict type system – values have statically derived types that they satisfy
- It is NOT the case that each value has a unique type: relation, not function
- Assume that global variables all satisfy their declared types
- Now want to know some fact -- e.g. what is the value of:

```
(v_bitvec->width
  (getrec "fld" (getarr 5 (gettuple 3 (getrec "fld2" (getglobal "myglob" env))))))
```

Type derivation example

Simplify:

```
(v_bitvec->width
  (getrec "fld" (getarr 5 (gettuple 3 (getrec "fld2" (getglobal "myglob" env))))))
```

Relevant rewrite rule -- find out width from type:

```
(implies (and (val-satisfies-type val type)
              (eq (type-kind type) :t_bitvec))
         (equal (v_bitvec->width val) (t_bitvec->width type)))
```

Type derivation example

Simplify:

```
(v_bitvec->width
  (getrec "fld" (getarr 5 (gettuple 3 (getrec "fld2" (getglobal "myglob" env))))))
```

Relevant rewrite rule -- find out width from type:

```
(implies (and (val-satisfies-type val type)
              (eq (type-kind type) :t_bitvec))
         (equal (v_bitvec->width val) (t_bitvec->width type)))
```

Type derivation example

Simplify:

```
(v_bitvec->width
 (getrec "fld" (getarr 5 (gettuple 3 (getrec "fld2" (getglobal "myglob" env))))))
```

Relevant rewrite rule -- find out width from type:

```
(implies (and (val-satisfies-type val type)
              (eq (type-kind type) :t_bitvec))
         (equal (v_bitvec->width val) (t_bitvec->width type)))
```

Type is a free variable – how to assign it?

Type derivation example

Simplify:

```
(v_bitvec->width
 (getrec "fld" (getarr 5 (gettuple 3 (getrec "fld2" (getglobal "myglob" env))))))
```

Relevant rewrite rule -- find out width from type:

```
(implies (and (val-satisfies-type val type)
              (eq (type-kind type) :t_bitvec))
         (equal (v_bitvec->width val) (t_bitvec->width type)))
```

Type is a free variable – how to assign it? Attempted rewrite:

```
(implies (and (val-satisfies-type val type)
              (eq (type-kind type) :t_rec)
              (equal typelook (assoc field (t_rec->fieldtypes type)))
              typelook)
         (val-satisfies-type (getrec field val) (cdr typelook)))
```

Type derivation example

Simplify:

```
(v_bitvec->width
  (getrec "fld" (getarr 5 (gettuple 3 (getrec "fld2" (getglobal "myglob" env))))))
```

Relevant rewrite rule -- find out width from type:

```
(implies (and (val-satisfies-type val type)
              (eq (type-kind type) :t_bitvec))
         (equal (v_bitvec->width val) (t_bitvec->width type)))
```

Type is a free variable – how to assign it when attempting to apply this rewrite?

```
(implies (and (val-satisfies-type val type)
              (eq (type-kind type) :t_rec)
              (equal typelook (assoc field (t_rec->fieldtypes type)))
              typelook)
         (val-satisfies-type (getrec field val) (cdr typelook)))
```

Type derivation example

Simplify:

```
(v_bitvec->width
  (getrec "fld" (getarr 5 (gettuple 3 (getrec "fld2" (getglobal "myglob" env))))))
```

Relevant rewrite rule -- find out width from type:

```
(implies (and (val-satisfies-type val type)
              (eq (type-kind type) :t_bitvec))
         (equal (v_bitvec->width val) (t_bitvec->width type)))
```

Type is a free variable – how to assign it when attempting to apply this rewrite?

```
(implies (and (val-satisfies-type val type)
              (eq (type-kind type) :t_rec)
              (equal typelook (assoc field (t_rec->fieldtypes type)))
              typelook)
         (val-satisfies-type (getrec field val) (cdr typelook)))
```

Base case: type of global lookup

Target term:

```
(v_bitvec->width
  (getrec "fld" (getarr 5 (gettuple 3 (getrec "fld2" (getglobal "myglob" env))))))
```

Finally get to innermost subexpression `(getglobal "myglob" env)`

Its type resolution rule:

```
(implies (and (fgl::match-assums (globals-satisfy-types env typectx))
              (equal look (assoc name typectx))
              look)
         (val-satisfies-type (getglobal name env) (cdr look)))
```

Here typectx is bound by a different mechanism (more like ACL2's): examine the current assumptions (path condition) to find a matching one. This rule will work if we have an assumption

```
(globals-satisfy-types env (my-global-types))
```

Replaying Abstract Counterexamples

Replaying Counterexamples of Abstractions

- FGL proofs usually end with a SAT check – UNSAT indicates proof, SAT indicates counterexample
- Satisfying assignment gives value to each Boolean variable
- But Boolean variables really stand for terms – need to map back to theorem variables to test whether the counterex really works
- Sometimes it's easy – simple destructors/accessors/lookups
- **But as soon as you stub out some nontrivial internal function, you can't really do this anymore ☹**

Abstract counterexample example

Ex.: function takes input 4-bit vector x, runs sqrt(x), does some other stuff

You stub out sqrt(x) -- it shouldn’t matter what exactly it produces, just another bitvector

Get a counterexample to your conjecture in terms of

x[0],..., x[3], sqrt(x)[0],...,sqrt(x)[3]

But it isn’t consistent – the Boolean assignment for the sqrt bits isn’t the result of sqrt applied to the x bits

Examining the counterexample

"But I just want to debug the counterexample and find out why it failed!"

- We could display it like a clause – terms associated with Boolean variables, negated or not
  - Could find out, e.g., sqrt result in counterexample is negative, which sends simulation down the wrong path and isn't possible – need to refine abstraction
  - But usually hundreds of big terms to look through
- Our approach instead: replay the (exact same) symbolic execution, but show/collect debugging info regarding the counterexample

Reference Counterexample + Path Condition

FGL tracks the current path condition purely as a Boolean expression – can tell whether the counterexample takes any given IF branch

Can build lots of debugging capabilities on this feature:

- Trace symbolic simulation of ASL function calls that are executed under the counterexample
- Print assignments that occur under the counterexample

Symbolic Simulation under No-Error Assumption

Symbolic Simulation under No-Error Assumption

- Often want to prove a property under an assumption that certain errors don't happen

```
(let ((res (interp ...)))
   (implies (not (member (result-error-code res) '(...)))
            (prop res)))
```

- Yet when symbolically simulating the run above, result from the interpreter must represent all the possible error conditions, before ruling them out

Interpreter redefinition under no-error assumption

- Typical interpreter structure – at any error, return immediately
  - Recursive calls always check for error results and return them
- Can eliminate these checks under assumption that errors don't occur

```
(equal (interp ...)
  (if ...
    (let ((rec-result (interp ...)))
      (if (is-error rec-result)
          rec-result
        ...))
   ...)
```

```
(implies (interp-no-error ...)
 (equal (interp ...)
   (if ...
     (let ((rec-result (interp ...)))
       ...)
    ...)
```

Propagating assumption to recursive calls

```
(equal (interp ...)
  (if ...
    (let ((rec-result (interp ...)))
      (if (is-error rec-result)
          rec-result
        ...))
   ...)
```

```
(implies (interp-no-error ...)
 (equal (interp ...)
   (if ...
     (let ((rec-result (interp ...)))
       ...)
    ...)
```

Not quite enough, because recursive call now doesn't have no-error assumption

Add a rewrite rule for every recursive call? Awkward and tons of backchaining:

```
(implies (and (interp-no-error ...)
              ...)
         (interp-no-error ...))
```

Propagating assumptions to recursive calls

```
(equal (interp ...)
  (if ...
    (let ((rec-result (interp ...)))
      (if (is-error rec-result)
          rec-result
        ...))
   ...)
```

```
(implies (interp-no-error ...)
 (equal (interp ...)
   (if ...
     (let ((rec-result
             (conditionalize
               __tmp-var-1
               (interp-no-error ...)
               (interp ...))))
       ...)
    ...))
```

Conditionalize semantics

```
(defun conditionalize (free-var test x)
  (if test x free-var))
```

FGL built-in: `(conditionalize free-var test x)` is rewritten as follows:

- Rewrite `test` yielding Boolean expression
- Rewrite `x` under the assumption of that Boolean expression
- Bind `free-var` to the result of that rewrite, return that same result
- Result: `x`, rewritten under the assumption that `test` is true.

Concluding thoughts

A verified proof engine gives confidence in experimental tweaks and new features

- This can go a long way toward tailoring it to your use cases
- Interaction, UI, debugging tools are super important in ITP, no matter how automatic you want things to be
- ACL2 ASL library on github

Ty!
Qs?

Backup

Preventing loops in non-transforming rewrite rules

Preventing loops when rewrite rules don’t change the target term

- Rewrite rule that checks what its target rewrites to, throws an error if it wasn’t successfully rewritten

```
(equal (foo x y)
       (let ((result (foo x y)))
         (fgl-prog2 (and (syntax-interp (is-call-of-foo result))
                         (fgl-error :msg "foo call no rewrite"))
                    result)))
```

- Rewrite rule that prints info about the target term before and after it’s rewritten

```
(equal (bar x y)
       (fgl-prog2
        (syntax-interp (cw "> bar x: ~x0 y: ~x1~%" x y))
        (let ((result (bar x y)))
         (fgl-prog2 ...
                    result))))
```

Examining the stack for function annotations

- Unfortunately both these rules loop
- However, we now have a workaround – examine the rewrite stack to see if your rule has already been applied to this target

```
(implies (and (bind-function-annotation annot 'foo equal (foo x y))
              (not (member-eq :check-foo annot)))
         (equal (foo x y)
                (let ((result (annotate (cons :check-foo annot) (foo x y))))
                  (fgl-prog2 (and (syntax-interp (is-call-of-foo result))
                                  (fgl-error :msg "foo call no rewrite"))
                             result))))
```

Symbolic tracing using a rewrite rule

Bonus – can store extra information in the annotation -- e.g.: call depth for tracing

```
2> eval_subprogram-*t "FPUnpack"
 3> eval_subprogram-*t "FPUnpackBase-1"
  4> eval_subprogram-*t "FPUnpackBase"
   5> eval_subprogram-*t "IsFeatureImplemented"
   <5 eval_subprogram-*t "IsFeatureImplemented"
   5> eval_subprogram-*t "UsingAArch32"
    6> eval_subprogram-*t "HaveAArch32"
     7> eval_subprogram-*t "IsFeatureImplemented"
     <7 eval_subprogram-*t "IsFeatureImplemented"
    <6 eval_subprogram-*t "HaveAArch32"
    6> eval_subprogram-*t "HaveAArch64"
     7> eval_subprogram-*t "IsFeatureImplemented"
     <7 eval_subprogram-*t "IsFeatureImplemented"
    <6 eval_subprogram-*t "HaveAArch64"
   <5 eval_subprogram-*t "UsingAArch32"
```