A verified runtime for a verified theorem prover

Magnus Myreen[1] and Jared Davis[2]

[1] University of Cambridge, UK
[2] Centaur Technology, USA

Two projects meet

My theorem prover is written in Lisp.
Can I try your verified Lisp?

Sure, try it.

Does your Lisp support ..., ... and ...?

No, but it could ...

Jared Davis

A self-verifying theorem prover

Milawa

Magnus Myreen

Verified Lisp implementations

verified **LISP** *on*
ARM, x86, PowerPC

Running Milawa

Milawa

verified **LISP**
ARM, x86, PowerPC
(TPHOLs 2009)

Jitawa: verified Lisp
with JIT compiler

hopelessly "toy"

Milawa's bootstrap proof:

- 4 gigabyte proof file:
  >500 million unique conses
- takes 16 hours to run on a state-of-the-art runtime (CCL)

Contribution:

- a new verified Lisp which is able to host the Milawa thm prover

Outline

Part 1: **Milawa**

Part 2: **Its new verified runtime**

Part 3: **Mini-demos, measurements**

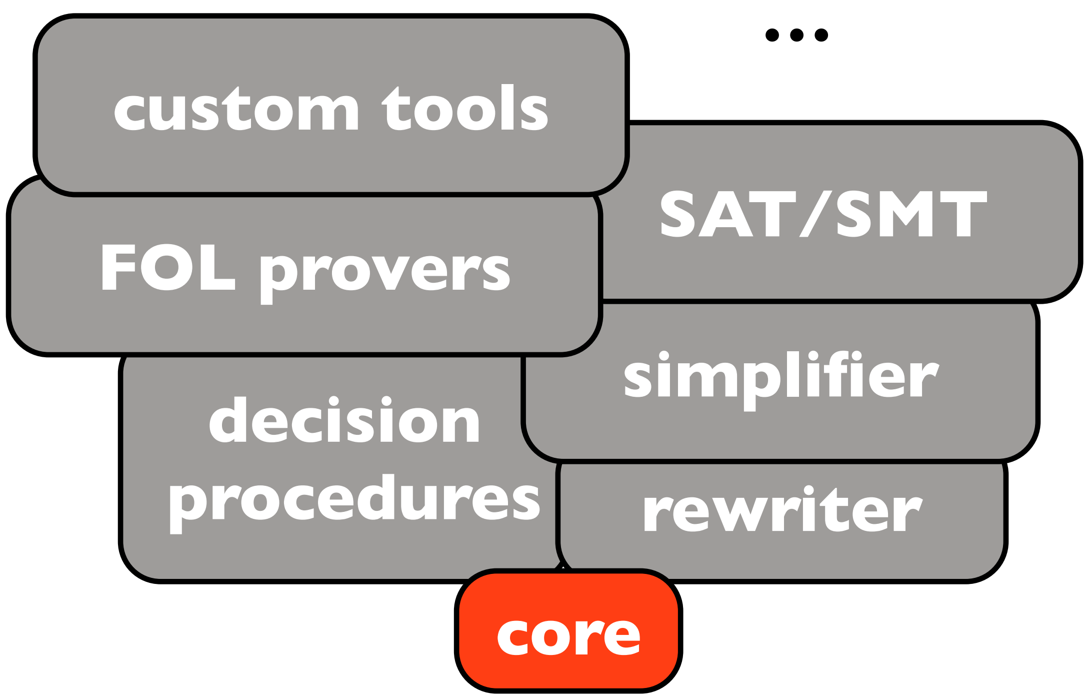
work by Jared Davis

A short introdution to Milawa

- Milawa is styled after theorem provers such as NQTHM and ACL2,
- has a small trusted logical kernel similar to LCF-style provers,
- ... but does not suffer the performance hit of LCF's fully expansive approach.

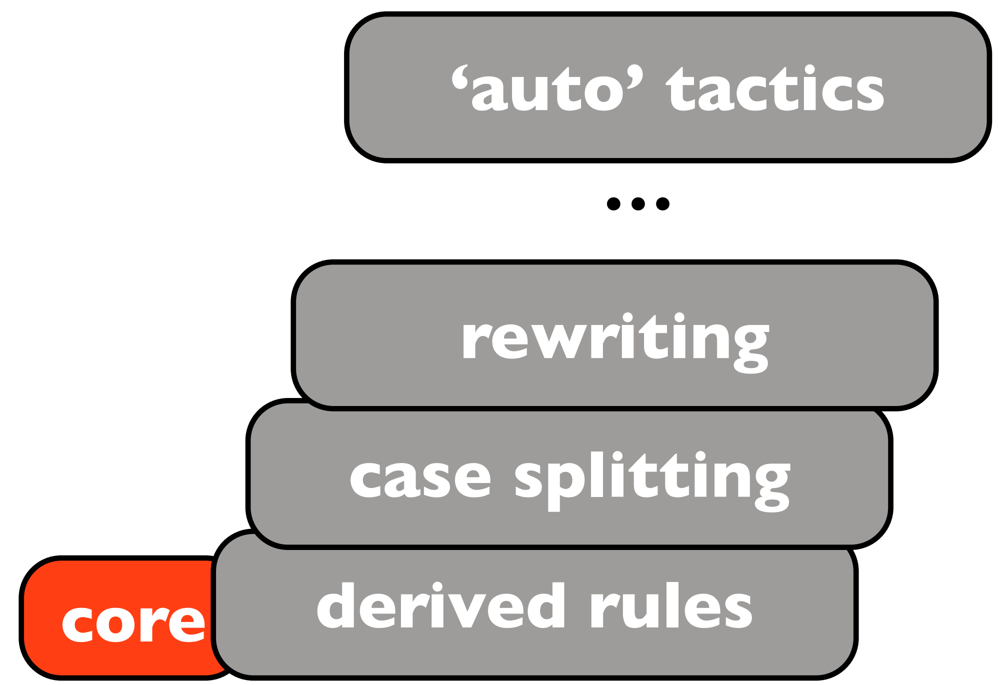
work by Jared Davis

Comparison with LCF approach



LCF-style approach

- all proofs pass through the core's primitive inferences
- extensions steer the core



the Milawa approach

- all proofs must pass the core
- the core can be reflectively extended at runtime

work by Jared Davis

Bootstrapping Milawa

Output from Milawa's bootstrap proof:

starts with very basic definitions and lemmas

```
(PRINT (1 VERIFY THEOREM-SUBSTITUTE-INTO-NOT-PEQUAL))
(PRINT (2 VERIFY THEOREM-NOT-T-OR-NOT-NIL))
(PRINT (3 DEFINE NOT))
(PRINT (4 VERIFY NOT))
(PRINT (5 DEFINE IF
(PRINT (6 VERIFY IF
(PRINT (7 VERIFY TH
...
(PRINT (4611 VERIFY |INSTALL-NEW-PROOFP-LEVEL2.PROOFP|))
(PRINT (4612 SWITCH |LEVEL2.PROOFP|))
(PRINT (4613 VERIFY |BUST-UP-LOGIC.FUNCTION-ARGS-EXPENSIVE|))
...
(PRINT (15685 VERIF
(PRINT (15686 SWITCH |LEVEL11.PROOFP|))
...
SUCCESS
```

up to this point the original core is used

this event switches to a new extended core

the extended core is used from now onwards

10 core extensions during bootstrap

work by Jared Davis

Milawa's core extensions

supports high-level tactics, similar to ACL2

level 11	Induction and other tactics
level 10	Conditional rewriting
level 9	Evaluation and unconditional rewriting
level 8	Audit trails (in
level 7	Case splitting
level 6	Factoring, split
level 5	Assumptions a
level 4	Miscellaneous
level 3	Rules about primitive functions
level 2	Propositional reasoning
core	Primitive proof checker

can only process primitive inferences, axioms

Soundness preserved:

each core extension is proved correct w.r.t. the current core before a switch event is allowed.

work by Jared Davis

Milawa's logic

Prop. Schema
$$\frac{}{\neg A \vee A}$$

Contraction
$$\frac{A \vee A}{A}$$

Expansion
$$\frac{A}{B \vee A}$$

Associativity
$$\frac{A \vee (B \vee C)}{(A \vee B) \vee C}$$

Cut
$$\frac{A \vee B \quad \neg A \vee C}{B \vee C}$$

Instantiation
$$\frac{A}{A/\sigma}$$

w.r.t. ordinals up to ε_0

Induction

Reflexivity Axiom
$$x = x$$

Equality Axiom
$$x_1 = y_1 \rightarrow x_2 = y_2 \rightarrow x_1 = x_2 \rightarrow y_1 = y_2$$

Referential Transparency
$$x_1 = y_1 \rightarrow \ldots \rightarrow x_n = y_n \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

Beta Reduction
$$((\lambda\, x \quad x \quad \beta)\, t \quad t\,) = \beta/[x \leftarrow t \quad x \leftarrow t\,]$$

evaluation of any lisp primitive applied to constants

Base Evaluation
e.g., $1+2 = 3$

Lisp Axioms
e.g., $consp(cons(x, y)) = t$

56 axioms describing properties of Lisp primitives

Sho67, KM98

work by Jared Davis

Trusting Milawa

Milawa is trustworthy if:

- logic is sound — next talk?
- core implements the logic correctly — future work
- runtime executes the core correctly — this talk!

If the above are proved, then Milawa could be "the most trustworthy theorem prover".

Outline

Part 1: **Milawa**

Part 2: **Its new verified runtime** ←

Part 3: **Mini-demos, measurements**

work by Jared Davis

Requirements on runtime

Milawa uses a subset of Common Lisp which

is for most part first-order pure functions over natural numbers, symbols and conses,

uses primitives: `cons car cdr consp natp symbolp equal + - < symbol-< if`

macros: `or and list let let* cond first second third fourth fifth`

and a simple form of lambda-applications.

(Lisp subset defined on later slide.)

work by Jared Davis

Requirements on runtime

...but Milawa also

- ~~uses destructive updates, hash tables~~
- ~~prints status messages, timing data~~
- ~~uses Common Lisp's checkpoints~~

} not necessary

- forces function compilation
- makes dynamic function calls
- can produce runtime errors

} runtime must support

(Lisp subset defined on later slide.)

Runtime must scale

Designed to scale:

- just-in-time compilation for speed
  - functions compile to native code
- target 64-bit x86 for heap capacity
  - space for 2^{31} (2 billion) cons cells (16 GB)
- efficient scannerless parsing + abbreviations
  - must cope with 4 gigabyte input
- graceful exits in all circumstances
  - allowed to run out of space, but must report it

Workflow

~30,000 lines of HOL4 scripts

1. specified input language: syntax & semantics
2. verified necessary algorithms, e.g.
   - compilation from source to bytecode
   - parsing and printing of s-expressions
   - copying (generational) garbage collection
3. proved refinements from algorithms to x86 code
4. plugged together to form read-eval-print loop

AST of input language

Example of semantics for macros:

$$\frac{(\mathsf{App}\ (\mathsf{PrimitiveFun}\ \mathsf{Car})\ [x], env, k, io) \xrightarrow{\mathsf{ev}} (ans, env', k', io')}{(\mathsf{First}\ x, env, k, io) \xrightarrow{\mathsf{ev}} (ans, env', k', io')}$$

	|	List ($term$ list)	(macro)
	|	Let (($string \times term$) list) $term$	(macro)
	|	LetStar (($string \times term$) list) $term$	(macro)
	|	Cond (($term \times term$) list)	(macro)
	|	First $term$ | Second $term$ | Third $term$	(macro)
	|	Fourth $term$ | Fifth $term$	(macro)
$func$	::=	Define | Print | Error | Funcall	
	|	PrimitiveFun $primitive$ | Fun $string$	
$primitive$	::=	Equal | Symbolp | SymbolLess	
	|	Consp | Cons | Car | Cdr |	
	|	Natp | Add | Sub | Less	

compile: AST $\longrightarrow$ bytecode list

bytecode	::=	Pop	pop one stack element
	|	PopN *num*	pop n stack elements
	|	PushVal *num*	push a constant number
	|	PushSym *string*	push a constant symbol
	|	LookupConst *num*	push the nth constant from system state
	|	Load *num*	push the nth stack element
	|	Store *num*	overwrite the nth stack element
	|	DataOp *primitive*	add, subtract, car, cons, . . .
	|	Jump *num*	jump to program point n
	|	JumpIfNil *num*	conditionally jump to n
	|	DynamicJump	jump to location given by stack top
	|	Call *num*	static function call (faster)
	|	DynamicCall	dynamic function call (slower)
	|	Return	return to calling function
	|	Fail	signal a runtime error
	|	Print	print an object to stdout
	|	Compile	compile a function definition

How do we get just-in-time compilation?

Treating code as data:

$$\forall p\ c\ q.\quad \{p\}\ c\ \{q\} \;=\; \{p * \textsf{code}\ c\}\ \emptyset\ \{q * \textsf{code}\ c\}$$

(POPL'10)

Solution:

- bytecode is represented by numbers in memory that <u>are</u> x86 machine code
- we prove that jumping to the memory location of the bytecode executes it

I/O and efficient parsing

Jitawa implements a read-eval-print loop:

Use of external C routines adds assumptions to proof:

- reading next string from stdin
- printing null-terminated string to stdout

An efficient s-expression parser (and printer) is proved, which deals with abbreviations:

```
(append (cons (cons a b) c)
        (cons (cons a b) c))
```

```
(append #1=(cons (cons a b) c)
        #1#)
```

Read-eval-print loop

- Result of reading lazily, writing eagerly
- Eval = compile then jump-to-compiled-code
- Specification: read-eval-print until end of input

$$\frac{\mathsf{is\_empty}\ (\mathsf{get\_input}\ io)}{(k, io) \xrightarrow{\mathsf{exec}} io}$$

$$\frac{\begin{array}{l}\neg\mathsf{is\_empty}\ (\mathsf{get\_input}\ io) \wedge \\ \mathsf{next\_sexp}\ (\mathsf{get\_input}\ io)) = (s, rest) \wedge \\ (\mathsf{sexp2term}\ s, [], k, \mathsf{set\_input}\ rest\ io) \xrightarrow{\mathsf{ev}} (ans, k', io') \wedge \\ (k', \mathsf{append\_to\_output}\ (\mathsf{sexp2string}\ ans)\ io') \xrightarrow{\mathsf{exec}} io''\end{array}}{(k, io) \xrightarrow{\mathsf{exec}} io''}$$

Correctness theorem

There must be enough memory and I/O assumptions must hold.

This machine-code Hoare triple holds only for terminating executions.

$$\{\,\mathsf{init\_state}\ io * \mathsf{pc}\ p * \langle \mathsf{terminates\_for}\ io \rangle\,\}$$

$$p : \mathsf{code\_for\_entire\_jitawa\_implementation}$$

list of numbers

$$\{\,\mathsf{error\_message} \vee \exists io'.\ \langle ([\,], io) \xrightarrow{\mathsf{exec}} io' \rangle * \mathsf{final\_state}\ io'\,\}$$

Each execution is allowed to fail with an error message.

If there is no error message, then the result is described by the high-level op. semantics.

Verified code

```
$ cat verified_code.s

    /*  Machine code automatically extracted from a HOL4 theorem.  */
    /*  The code consists of 7423 instructions (31840 bytes).      */

     .byte   0x48, 0x8B, 0x5F, 0x18
     .byte   0x4C, 0x8B, 0x7F, 0x10
     .byte   0x48, 0x8B, 0x47, 0x20
     .byte   0x48, 0x8B, 0x4F, 0x28
     .byte   0x48, 0x8B, 0x57, 0x08
     .byte   0x48, 0x8B, 0x37
```

How is this verified binary produced?

Demo: proof-producing synthesis (TPHOLs'09)

```
     .byte   0xC7, 0x00, 0x02, 0x54, 0x06, 0x51
     .byte   0x48, 0x83, 0xC0, 0x04
     ...
```

Outline

Part 1: **Milawa**

Part 2: **Its new verified runtime**

Part 3: **Mini-demos, measurements** ←

A short demo:

Jitawa — a verified runtime for Milawa

Running Milawa on Jitawa

Running Milawa's 4-gigabyte booststrap process:

CCL	16 hours	
SBCL	22 hours	
Jitawa	128 hours	(8x slower than CCL)

Jitawa's compiler performs almost no optimisations.

Parsing the 4 gigabyte input:

CCL	716 seconds	(9x slower than Jitawa)
Jitawa	79 seconds	

Quirky behaviour

DEMO

Jitawa mimics an interpreter's behaviour

- to hide the fact that compilation occurs
- to keep semantics as simple as possible
- to facilitate future work (e.g. verify Milawa's core)

Consequences:

- compiler must turn undefined functions, bad arity and unknown variables into runtime checks/fails.
- mutual recursion is free!

Q&A

Did the verified runtime work immediately?

No, there were bugs in the 64-bit x86 model.

What is assumed?

x86 model, C wrapper, OS, hardware

May I use the verified runtime Jitawa?

yes! Link on next slide...

Questions?

Website: http://www.cl.cam.ac.uk/~mom22/jitawa/

Jitawa ≈ "jittaava"
which is the active present
participle form of the verb "jitata"

Finnish for
"to JIT compile"