Building Verified Program Analyzers in Coq

Lecture 4: A verified value analysis for CompCert

David Pichardie - INRIA Rennes / Harvard University

The CompCert verified C compiler

Compiler built and proved by Xavier Leroy et al.

Slides largely inspired by Leroy's own material

critical_prog.ppc

	.const		L101:			L184:		
	,align	2		movsd	40(Nesp), Xxmm6		mov1	Nebx, O(Nesp)
stri	inglit_1:			movsd	32(Nesp), Xxmm7		cell	_test
	.ascii	"integr(square, 0.0, 1.0,		mulsd	Xxmn7 Xxmm6		subl	\$8, Xesp
	.text			movsd	Xom6, 40(Xesp)		fstpl	O(Nesp)
	.align	4		fldl	40(Kesp)		movsd	O(Nesp), NormO
_square	12			movl	20(Mesp), Nebx		odd1	\$8, Wesp
	sub1	\$12, %esp		movl			leal	stringlit_1, Mean
	lecl	16(Mesp), Medx		oddl	24(Mesp), Mesi \$60, Mesp		mov1	Neax, O(Nesp)
	mov1	Medx, O(Mesp)			soo, sesp		movl	Nebx, 4(Nesp)
	mov1	@(Nesp), Nedx		ret			movsd	Xxmm8, 8(Xesp)
	movsid	@(Nedx), Norm@		.text			cell	_printf
	mulsd	Norma, Norma		.align			xorl	Neax, Neax
	sub1	\$8, Xesp		.globl	_test		movt	28(Wesp), Mebx
	movsd	Monnel, @(Mesp)	_test:		A		odd1	\$44, Xesp
	กก	Ø(Nesp)		subl	\$44, %esp		ret	and wear
	oddl	\$8, Xesp		leal	48(Mesp), Medx			
	oddl	\$12, Xesp		movl.	Medx, 24(Mesp)			
	ret			mov1	Mebx, 36(Mesp)			
	.text			movl	24(Mesp), Medx			
d antices	.align			movl	O(Nedx), Nedx			
_integr		\$60, Xesp		leal	_square, %ebx			
	legl			xorpd	Xxmm2, Xxmm2 # +0.0			
	movl	64(Mesp), Medx Medx, B(Mesp)		movsd	L182, %xmm1 # 1			
	movl	Mebx, 28(Mesp)		movl	Metrx, B(Mesp)			
	mov1	Mesi, 24(Mesp)		movsd	Xxmn2, 4(Nesp)			
	movl	B(Nesp), Nedx		movsd	Xxmm1, 12(Xesp)			
	movl	O(Nedx), Mesi		movl	Seax, 20(Sesp)			
	movl	B(Nesp), Nedx		call	_integr			
	movsd	4(Neds), Xarmi		subl	\$8, Xesp			
	novsd	Xormi, 48(Nesp)		fstpl	0(Xesp)			
	mov1	B(Nesp), Neds[]		movsd	O(Nesp), Normal			
	movad	12(Nedx), North		oddl	\$8, Xesp			
	movsd	Normi, 32(Nesp)		subl	\$8, Xesp			
	mov1	B(Nesp), Nedx		movsd	Skone, @(Nesp)			
	mov1	20(Medx), Nebx		fldl	O(Nesp)			
	novsd	32(Nesp), %xmm6		oddl	\$8, Nesp			
	movsd	48(Nesp), %xmm7		movl	36(Wesp), Nebx			
	subsd	Soren7 Noren6		odd1	\$44, Xesp			
	movsd	%omm6, 32(%esp)		net				
	cvtsiZs/	d Nebx, Namm1		.const.	data			
	movsd	32(Mesp), Xamm6		.cliph				
	divsd	Norm1, Norm6	L102:	.quad	8x3ff0000000000000			
		Xxmm6, 32(Xesp)		.text				
		Xorm6, Xorm6 # +0.0		.align	4			
	novsd	Xxmm6, 40(Xesp)		.globl				
1198:			moin:					
	cmpl	\$0, Xebx		subl	\$44, %esp			
	jle	L101		leal	48(Wesp), Nedx			
	movsd	48(Nesp), %xmm6		movl	Nedx, 16(Nesp)			
	movsd	Monné, @(Mesp)		movi	Nebx, 28(Nesp)			
	coll	Mes-1		movi				
	subl	\$8, Xesp			16(Mesp), Medx			
	fstpl	Ø(Xesp)		movl	8(Nedx), Nebx			
	movsd	@(Xesp), Xonn0		movl	16(Mesp), Medx			
	oddl	\$8, Xesp		movl	4(Nedx), Nedx			
	movsd	40(Nesp), Xxmm6		cmpl	\$2, Xebx			
	oddsd	Xored, Xored		jge	L183			
	movsd	Xxmb, 40(Xesp)		movl	\$1000000, Metx			
	lesl	-1(Nebx), Nebx		jmp	L184			
	movsd	48(Nesp), Xamb	L103:					
	movsd	32(Mesp), Monn7		movl	4(Neax), Neax			
	oddsd	Xorm7, Xorm6		movl.	Neax, 8(Nesp)			
	movsd	Xorm6, 48(Xesp) L100		call	_atoi			
	300	1.100		movl	Seax, Sebx			

Critical embedded software

Critical embedded software

High degree of assurance is requiredis the program critical_prog.ppc safe ?

	.const		L101:			L184:		
	.align	2		novsd	40(Nesp), Xonnó		movl	Nebx, @(Nesp)
stria				novsd	32(Nesp), %om7		cell	_test
	.ascii	"integr(square, 0.0, 1.0,		mulsd			subl	\$8, Xesp
	.text				Xonn7, Xonn6		fstpl	O(Nesp)
	.align	4		movsd	Xom6, 40(Xesp)		movsd	O(Nesp), Norma
square:				fldl	40(Xesp)		oddl	\$8, %esp
	subl	\$12, Wesp		movl	20(Mesp), Mebx		legt	stringlit_1, %
	lecl	16(Mesp), Medx		movl	24(Mesp), Mesi		movt	Neax, @(Nesp)
	mov1	Medx, 0(Mesp)		oddl	\$60, %esp		movt	Nebx, 4(Nesp)
	mov1	@(Nesp), Nedx		ret			movsd	Xxmm8, 8(Xesp)
	movsd	@(Nedx), Norm@		.text			cell	_printf
	mulsd	Norma, Norma		.align			xorl	Neax, Neax
	subl	\$8, Xesp		.globl	_cest		movt	28(Nesp), Nebx
	novsd	Xxxxx0, @(Xesp)	_test:		444 Marca		odd1	\$44, Xesp
	na	Ø(Nesp)		subl	\$44, %esp		net	Prof. Mark
	oddl	\$8, Xesp		leal	48(Mesp), Medx			
	oddl	\$12, Mesp		movl	Medx, 24(Mesp)			
	ret			movl,	Xebx, 36(Mesp)			
	.text			mov1	24(Kesp), Medx			
Interes	.align			movl	O(Nedx), Nedx			
_integr:	subl	\$60, Xesp		leal	_square, %ebx			
	lesl	64(Nesp), Nedx		xorpd	Xxrm2, Xxrm2 # +0.0			
	mov1	Nedx, B(Nesp)		movsd	L182, %xm1 # 1			
	movl	Mebx, 20(Mesp)		mov1	Metrx, B(Mesp)			
	mov1	Mesi, 24(Nesp)		movsd	Xxmn2, 4(Xesp)			
	mov1	8(Nesp), Nedx		movsd	Xxml, 12(Xesp)			
	mov1	O(Nedx), Nesi		movl	Seax, 20(Sesp)			
	mov1	8(Nesp), Nedx		call	_integr			
	movsd	4(Nedx), Xxmm6		subl	\$8, Xesp			
	novsd	Xxmm6, 48(Xesp)		fstpl	0(Xesp)			
	mov1	8(Nesp), Nedx		movsd	B(Nesp), %xmm8			
	movid	12(Nedx), %xmm6		oddl	\$8, Xesp			
	movsd	Normó, 32(Nesp)		subl	S8, Xesp			
	mov1	B(Nesp), Nedx		movsd	%xmm0, @(%esp)			
	mov1	20(Medx), Mebx		fldl	B(Kesp)			
	movsd	32(Nesp), %xmm6		oddl	S8, Xesp			
	movsd	48(Nesp), %xmm7		movl	36(Mesp), Mebx			
	subsd	Norm5, Norm6 Norm6, 32(Nesp)		oddl	\$44, %esp			
		d Nebx, Nami		ret				
	movid	32(Mesp), Xamm6		.const.	-			
	divsd	Now1, Now6		.align				
		Normi, 32(Nesp)	L102:	.quad	8x3ff000000000000000			
	xorpd	Xorm6, Xorm6 # +0.0		.text				
	novsd	Xorm6, 40(Xesp)		.align				
L100:				.globl	_main			
	cmpl	\$8, Xebx	_moin:					
	jle	L101		subl	\$44, %esp			
	movid	48(Nesp), %xmm6		leal	48(Wesp), Wedx			
	novsd	Monno, @(Nesp)		movl	Nedx, 16(Nesp)			
	call	*Xes1		movl	Mebx, 28(Mesp)			
	subl	\$8, Xesp		mov1	16(Mesp), Medx			
	fstpl	0(Nesp)		movl	O(Nedx), Nebx			
	movsd	@(Nesp), Norm@		movl	16(Mesp), Medx			
	oddl	\$8, Xesp		movl.	4(Nedx), Nedx			
	movsd	40(Nesp), Xxmm6		cmpl	\$2, Xebx			
	oddsd	Xormil, Xormi		jge	L183			
	movsd	Xxmmb, 48(Xesp)		movl	\$1000000, Metx			
	leol	-1(Nebx), Nebx		jmp	L184			
	movsd	48(Xesp), Xxm6	L103:					
	movsd	32(Xesp), Xom7		movl	4(Neax), Neax			
	oddsd	Xorm7, Xorm6		movl.	Neax, 8(Nesp)			
	movsd .	Xorm6, 48(Xesp) L100		call	_atoi			
	jmp	1.100		movl	Seax, Sebx			

critical_prog.ppc

Critical embedded software

High degree of assurance is required

• is the program critical_prog.ppc safe ?

2 options:

 qualify the PPC program as if hand-written (intensive testing, painful manual review...)

 qualify the program at the source level (static analysis, model checking, or program proof)

	const		101:			L184:		
	align.	2		novsd	40(Nesp), Xonn6		movl	Metox, @(Mesp)
string				movsd	32(Wesp), %xmm7		cell	_test
	ascii	"integr(square, 0.0, 1.0,		mulsd	Xxmn7 Xxmm6		subl	\$8, Wesp
	text			novsd	Xom6, 40(Xesp)		fstpl	O(Xesp)
	align :	4		fldl	40(%esp)		movsd	O(Nesp), Norma
_square:							oddl	\$8, %esp
	wb1	\$12, %esp		mov1	20(Mesp), Mebx			
	leci	16(Mesp), Medx		mov1	24(Mesp), Mesi		leal	stringlit_1, Xee
	lvov1	Nedx, O(Nesp)		oddl.	\$60, %esp		movl	Neax, O(Nesp)
	lov1	O(Nesp), Nedx		ret			movl	Mebx, 4(Mesp)
	bevos	@(Neds), Xarm0		.text			movsd	Some, 8(Nesp)
	ulsd	Xored Xored		.align	4		coll	_printf
	sub1	\$8, Xesp		.globl			xorl	Secx, Secx
	bevor		_test:				movl	28(Wesp), Webx
		Korn0, 0(Xesp)		subl	\$44, Xesp		odd1	\$44, Xesp
	ากก	O(Nesp)					net	
	oddl	\$8, Xesp		leal	48(Mesp), Medx			
	oddl	\$12, %esp		movl.	Medx, 24(Mesp)			
	net			movl.	Mebx, 36(Mesp)			
	text			movl.	24(Mesp), Medx			
	align.	4		movl	O(Nedx), Nedx			
_integr:				leal	_square, %ebx			
1	wb1	\$60, Xesp		xorpd	Xxmm2, Xxmm2 # +0.0			
1	lesl	64(Mesp), Nedx		movsd	L182, Xam1 # 1			
	nov1	Nedx, 8(Nesp)		movl				
	row1	Mebx, 20(Nesp)			Netrx, @(Nesp)			
	lvor	Mesi, 24(Mesp)		movsd	Xxm2, 4(%esp)			
	nov1	B(Nesp), Nedx		movsd	Kunn1, 12(Xesp)			
	rov1	O(Nedx), Mesi		movl	Neax, 20(Nesp)			
	lv0	B(Nesp), Nedx		call	_integr			
	novsd	4(Neds), Xarmi		subl	\$8, Xesp			
	tovid	Xum6, 48(Xesp)		fstpl	O(Xesp)			
	ov1	B(Nesp), Neds[]		movsd	O(Nesp), %xmm8			
	bevor	12(Nedx), Xamb		odd1	\$8, Xesp			
	tovsd	Kormi, 32(Kesp)		subl	\$8, Xesp			
				movsd	Kame, @(Mesp)			
	l vo	8(Nesp), Nedx 20(Nestr) Netry		fldl				
	NOV1	20(Nedx), Nebx			O(Nesp)			
	rovsd	32(Xesp), Xxmm6		oddl	\$8, Nesp			
	novsd	48(Nesp), %xmm7		movl	36(%esp), %ebx			
	subsid	Sorm7, Sorm6		oddl	\$44, %esp			
		Norm6, 32(Nesp)		ret				
		Nebx, Namm1		.const,	data			
	novsd	32(Mesp), Xxmm6		.align	3			
1	divsd	Norm1, Norm6	102:	.quad				
		Norm6, 32(Nesp)		.text				
3	corpd	Xorm6, Xom6 # +0.0		.align	4			
	lovsd	Xorm6, 40(Xesp)						
L180:			malant	.globl	-market fri			
	Igno	\$0, Nebx	_moin:					
	le	L101		subl	\$44, %esp			
	novsd	48(Nesp), Xxmm6		leal	48(Wesp), Wedx			
	bevor	Xormó, Ø(Xesp)		mov1	Nedx, 16(Nesp)			
	1100	"Xesi		mov1	Nebx, 28(Nesp)			
	ubl	\$8, Xesp		mov1	16(Nesp), Nedx			
	fstpl	0(Neso)		mov1	O(Nedx), Nebx			
				movl	16(Mesp), Nedx			
	lovsd	@(Nesp), Norm@						
	odd1	\$8, Nesp		movl	4(Nedx), Nedx			
		40(Wesp), %xmm6		cmpl	\$2, Xetx			
		Xoren0, Xoren6		jge	L183			
	novisd	Xxmm6, 48(Xesp)		movl	\$1000000, %ebx			
	legl	-1(Nebx), Nebx		jmp	L184			
	novsd	48(Nesp), Xxmm6	L103:					
	lovsd	32(Mesp), Monn7		movl	4(Neax), Neax			
	oddsd	Xierm7, Xierm6		mov1	Neax, @(Nesp)			
	bavon	Xorm6, 48(Xesp)		coll	_atoi			
	mp	L100		movl	Xeax, Xebx			

Critical embedded software

High degree of assurance is required

• is the program critical_prog.ppc safe ?

2 options:

 qualify the PPC program as if hand-written (intensive testing, painful manual review...)

 qualify the program at the source level (static analysis, model checking, or program proof)

2nd option is preferred in practice

- can you trust your compiler ?
- this talk: apply formal verification techniques to the compiler itself !

	.const		L101:			L184:		
	,align			movsd	40(Nesp), Xionn6		movl	Nebx, @(Nesp)
stri	nglit_1:			movsd	32(Nesp), %xmm7		call	_test
	.ascii	"integr(square, 0.0, 1.0,		mulsd	Xxmn7, Xxmm6		subl	\$8, %esp
	.text			movsd	Xxmm6, 48(Xesp)		fstpl	O(Kesp)
	.align	4		FLdL	40(Nesp)		movsd	O(Nesp), Norma
square		413 Free		mov1	20(Mesp), Mebx		oddl	\$8, %esp
	subl	\$12, Nesp		mov1	24(Mesp), Mesi		leal	stringlit_1, Meax
	lecl mov1	16(Mesp), Medx		oddl	\$60, %esp		movl	Meax, O(Mesp)
	mov1	Nedx, 0(Nesp) 0(Nesp), Nedx		net			movl	Nebx, 4(Nesp)
	movid	@(Neds), Norm8		.text			movsd	Xxmm0, 8(Nesp)
	mulsd	Norme, Norme		.align	4		coll	_printf
	subl	\$8, Xesp		.globl			xorl	Neax, Neax
	novsd	Morriel, @(Mesp)	_test:				movt	28(Wesp), Mebx
	na	@(Neso)		subl	\$44, %esp		odd1	\$44, %esp
	oddl	\$8, Xesp		leal	48(Mesp), Nedx		net	
	oddl	\$12, Mesp		mov1	Nedx, 24(Nesp)			
	ret			movl	Nebx, 36(Mesp)			
	.text			movl	24(Mesp), Medx			
	.align	4		movi	O(Nedx), Neux			
integr				leal	_square, %ebx			
	subl	\$60, Xesp		xorpd	Xxmm2, Xxmm2 # +0.0			
	lesl	64(Nesp), Nedx		movsd	L182, %xmm1 # 1			
	mov1	Nedx, 8(Nesp)		movt	Nebx, @(Nesp)			
	mov1	Mebx, 20(Mesp)		movsd	Xxmn2, 4(Nesp)			
	mov1	Mesi, 24(Mesp)		movsd	Kunn1, 12(Nesp)			
	mov1	B(Nesp), Nedx		movl	Neax, 20(Nesp)			
	movl	O(Nedx), Nesi		call	_integr			
	movl	B(Nesp), Nedx		subl	\$8, Xesp			
	movsd	4(Nedx), Xxrm6		fstpl	O(Nesp)			
	movsd	Xxmm6, 48(Xeso)		movsd	O(Nesp), Xxmm0			
	mov1 movsd	8(Nesp), Nedx[] 12(Nedx), Norm6		odd1	\$8, Xesp			
	movsd	Normi, 32(Ness)		subl	\$8, Xesp			
	mov1	B(Nesp), Nedx		movsd	Kame, @(Nesp)			
	mov1	20(Medx), Mebx		fldl	O(Nesp)			
	novsd	32(Mesp), Xonn6		oddl	\$8, Xesp			
	movsd	48(Nesp), %xmm7		movl	36(Nesp), Nebx			
	subsd	Soren7 Same6		oddl	\$44, %esp			
	novsd	Norm6, 32(Nesp)		ret	and weat			
	cvtsiZs	d Nebx, Norml		.const.	data			
	movsd	32(Mesp), Xarm6		.align				
	divsd	Norm1 Norm6	192:	.quad	8x3ff0000000000000			
	movsd	Norm6, 32(Nesp)		.text				
	xorpd	Xorm6, Xorm6 # +0.0		.align	4			
	movsd	Xorm6, 40(Xesp)		.globl				
100:			moin:	-91001	Party Party			
	cmpl	S0, Nebx	- Martinitz	subl	\$44, %esp			
	jle	L101		leal	48(Nesp), Nedx			
	movid	48(Kesp), Xonn6		movl	Nedx, 16(Nesp)			
	movsd	Monn6, @(Nesp)		mov1	Xebx, 28(Xesp)			
	coll	Xesi		movl	16(Mesp), Medx			
	subl	\$8, Xesp		movi	B(Medx), Mebx			
	fstpl	O(Nesp)		movi	16(Nesp), Nedx			
	oddl	@(Nesp), Xxmm0 \$8, Xesp		movi	4(Nedx), Nedx			
	movsd	40(Kesp), Xxmm6		cmpl	\$2, Netry			
	oddsd	Sorn0, Sonn6			L183			
	movid	Xxmm6, 40(Xesp)		jge				
	leol	-1(Nebx), Nebx		movl	\$18888888, Mebx			
	movsd	48(Nesp), Xonn6	L103:	Jub	L184			
	movsd	32(Mesp), More?	1103:	man d	Alterna Marris			
	oddsd	Norm7 Norm6		movl	4(Xeax), Xeax			
	movisid	Xormi, 48(Xesp)		coll	Neax, @(Nesp)			
				COLL	_atoi			
	300	L100		movl.	Neax, Nebx			

Xuejun Yang Yang Chen Eric Eide John Regehr, *Finding and Understanding Bugs in C Compilers*, PLDI 2011

Xuejun Yang Yang Chen Eric Eide John Regehr, *Finding and Understanding Bugs in C Compilers*, PLDI 2011

We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. [...]

Xuejun Yang Yang Chen Eric Eide John Regehr, *Finding and Understanding Bugs in C Compilers*, PLDI 2011

We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. [...]

We found and reported hundreds of previously unknown bugs. [...]

Xuejun Yang Yang Chen Eric Eide John Regehr, *Finding and Understanding Bugs in C Compilers*, PLDI 2011

We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. [...]

We found and reported hundreds of previously unknown bugs. [...]

Many of the bugs we found cause a compiler to emit incorrect code without any warning. 25 of the bugs we reported against GCC were classified as release-blocking.

Xuejun Yang Yang Chen Eric Eide John Regehr, *Finding and Understanding Bugs in C Compilers*, PLDI 2011

We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. [...]

We found and reported hundreds of previously unknown bugs. [...]

Many of the bugs we found cause a compiler to emit incorrect code without any warning. 25 of the bugs we reported against GCC were classified as release-blocking.

Most of the bugs we found in GCC were in the middle end: the machine- independent optimizers.

Xuejun Yang Yang Chen Eric Eide John Regehr, *Finding and Understanding Bugs in C Compilers*, PLDI 2011

We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. [...]

We found and reported hundreds of previously unknown bugs. [...]

Many of the bugs we found cause a compiler to emit incorrect code without any warning. 25 of the bugs we reported against GCC were classified as release-blocking.

Most of the bugs we found in GCC were in the middle end: the machine- independent optimizers.

Optimizing compilers rely on complex static analyses!

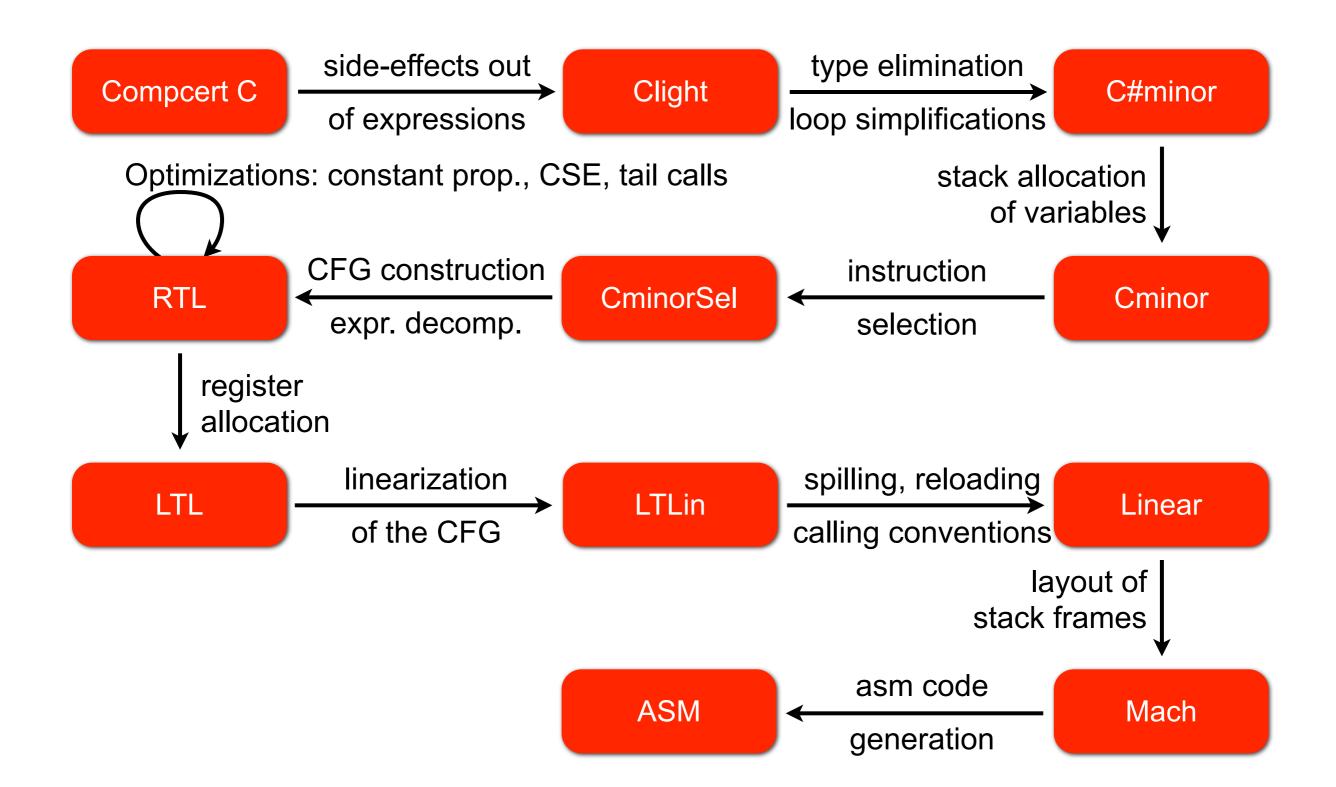
The CompCert project X. Leroy, S. Blazy et al.

Develop and prove correct a realistic compiler, usable for critical embedded software.

- Source language: a very large subset of C.
- Target language: PowerPC/ARM/x86 assembly.
- Generates reasonably compact and fast code
 ⇒ careful code generation; some optimisations.

Note: compiler written from scratch, along with its proof; not trying to prove an existing compiler (otherwise see Zdancewic et al's Verified LLVM project).

The formally verified part of the compiler



Formally verified in Coq

After 50 000 lines of Coq and 4 person.years of effort

```
Theorem transf_c_program_is_refinement:
  ∀ p tp,
  transf_c_program p = OK tp →
  (∀ beh, exec_C_program p beh → not_wrong beh) →
  (∀ beh, exec_asm_program tp beh → exec_C_program p beh).
```

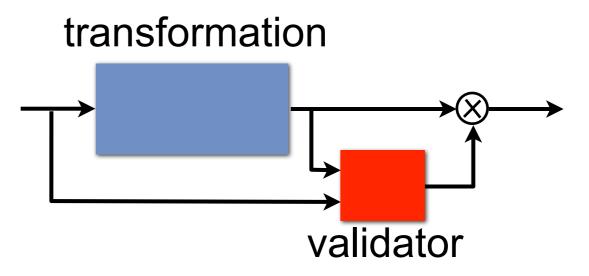
Behaviors beh = termination / divergence / going wrong + trace of I/O operations (syscalls, volatile accesses).

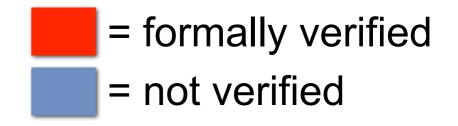
Verified transformation

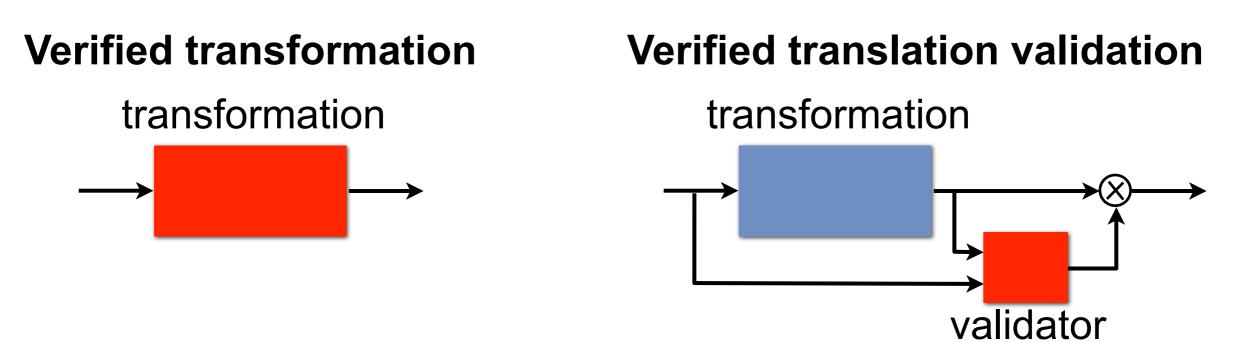


Verified transformation transformation \longrightarrow

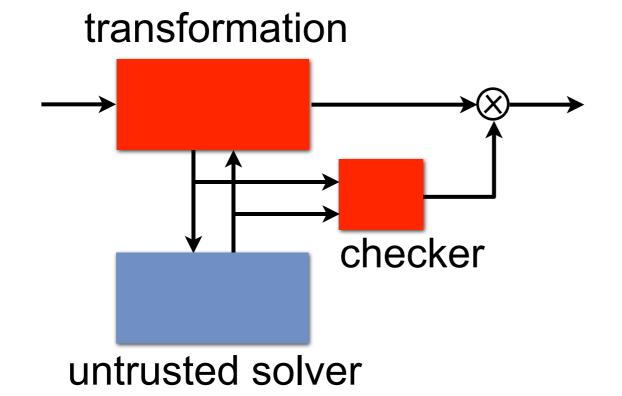
Verified translation validation

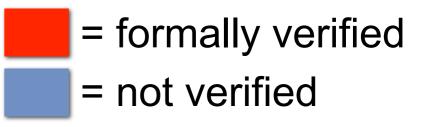






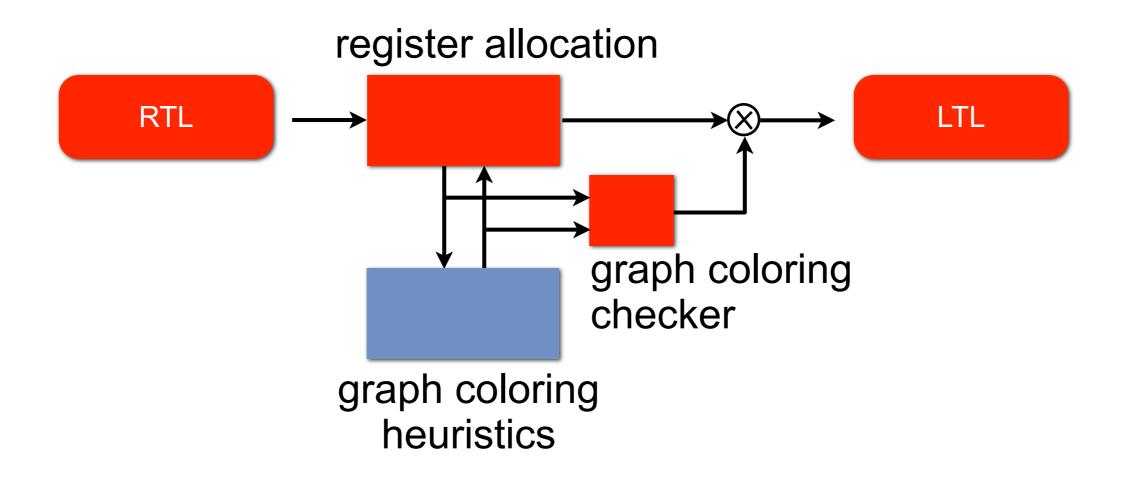
External solver with verified validation





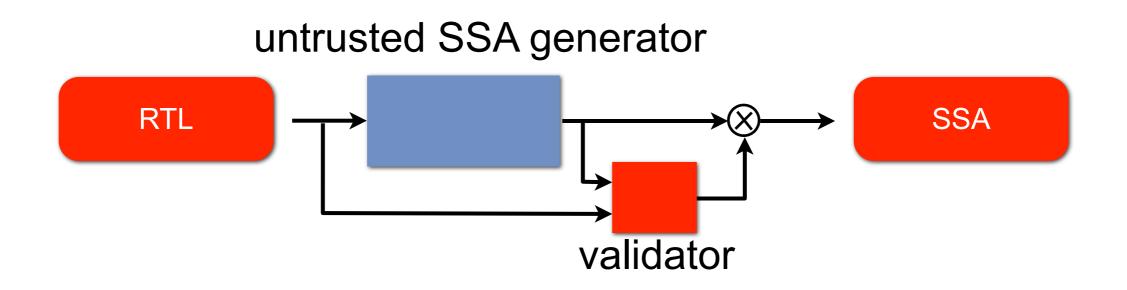
External solver with verified validation

Example: register allocation



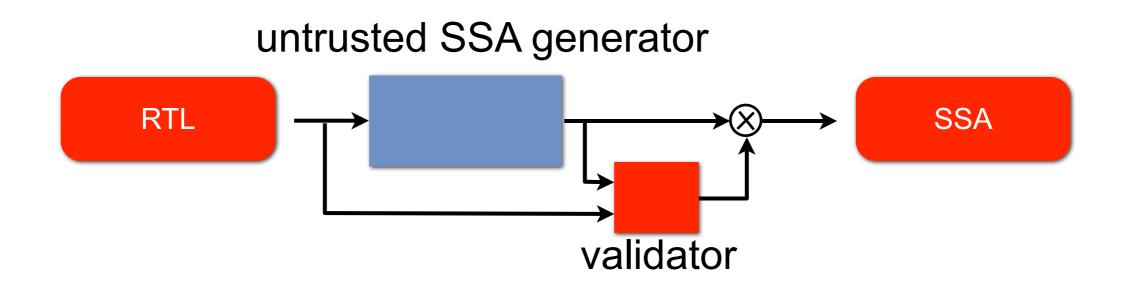
Verified translation validation

Example: SSA generation (in CompCert SSA extension)



Verified translation validation

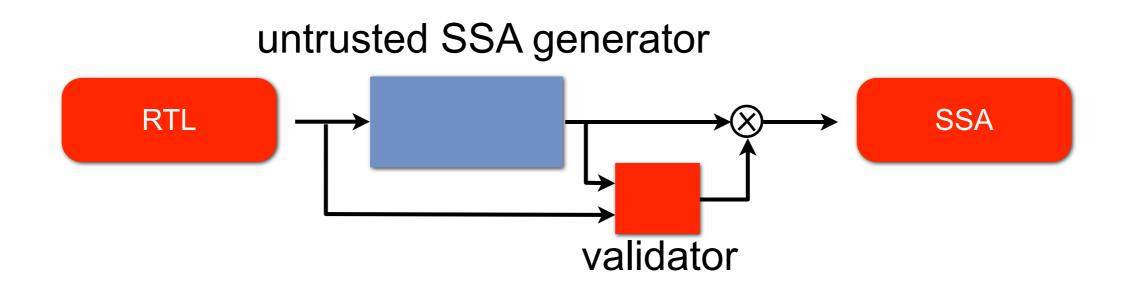
Example: SSA generation (in CompCert SSA extension)



The untrusted generator can rely on advanced graph algorithms as Lengauer and Tarjan's dominator tree construction and frontier dominance computation.

Verified translation validation

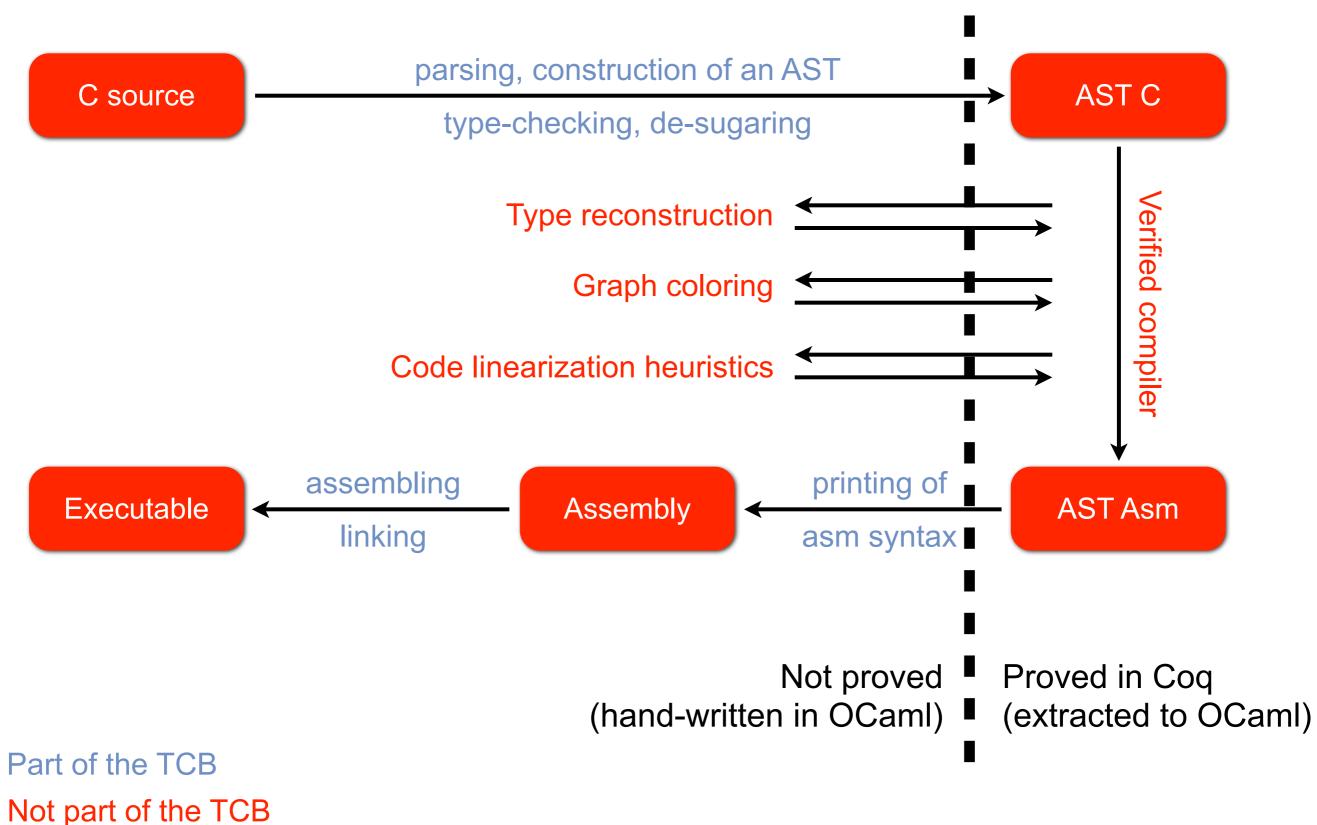
Example: SSA generation (in CompCert SSA extension)



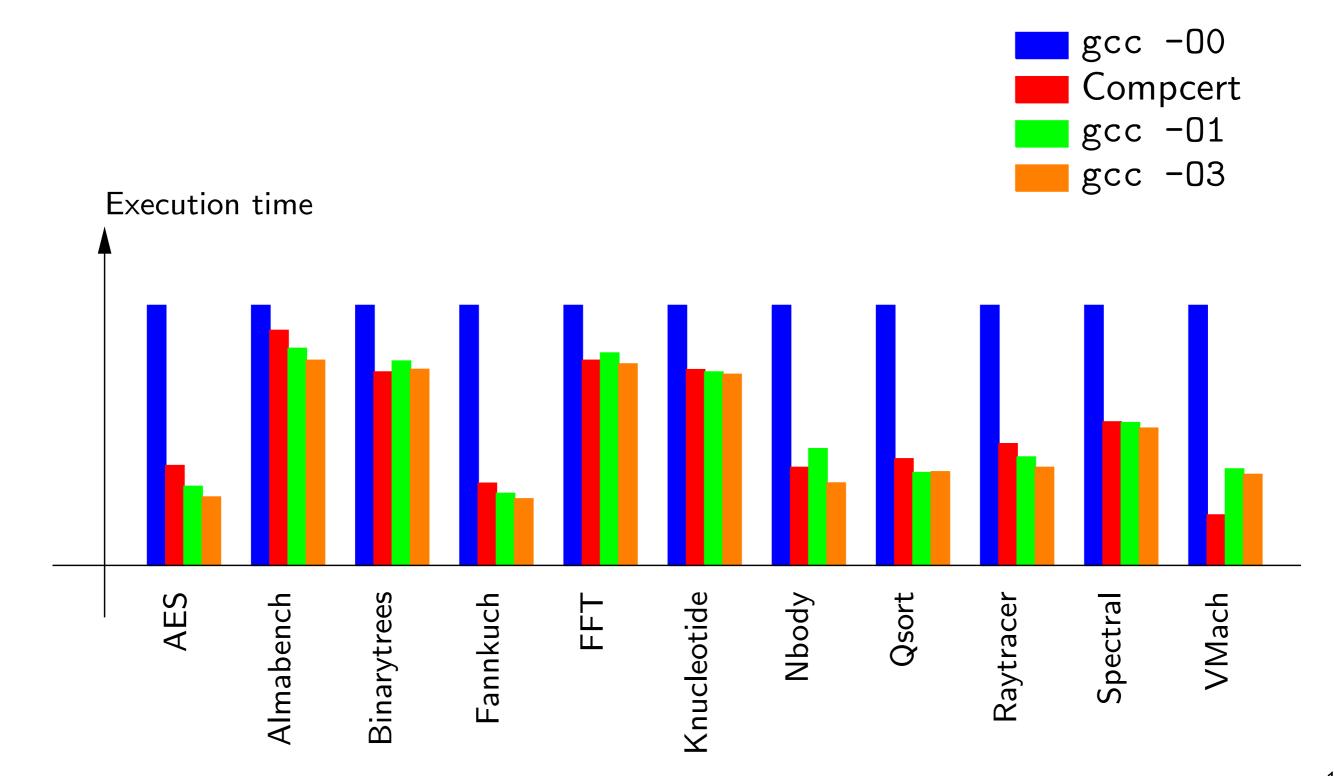
The untrusted generator can rely on advanced graph algorithms as Lengauer and Tarjan's dominator tree construction and frontier dominance computation.

We prove the validator is *complete* with respect to this family of algorithms.

The whole CompCert compiler



Performance of generated code (On a PowerPC G5 processor)



Conclusions

The formal verification of realistic compilers is feasible.

(Within the limitations of contemporary proof assistants)

Much work remains:

- Shrinking the TCB (e.g. verified parsing, validated assembling & linking).
- More optimizations (see CompCert SSA).
- Front-ends for other languages
- Concurrency (see Sevcik et al's CompCert TSO and Appel and al's Verified Software Toolchain).
- Connections with source-level verification (ongoing french project on a *verified* C static analyzer)

Formal Verification of a C Value Analysis

Sandrine Blazy, Vincent Laporte, Andre Maroneze, and David Pichardie, to be presented at the 20th Static Analysis Symposium 2013

This work is part of the common Verasco project between

Airbus

INRIA Paris Rocquencourt (Gallium, Abstraction)

INRIA Saclay (Toccata)

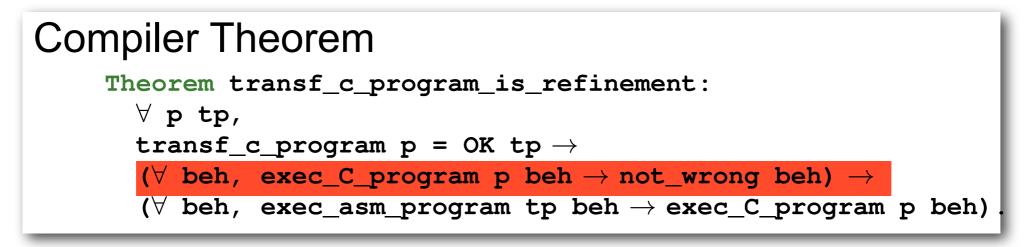
Université Rennes I (Celtique)

VERIMAG



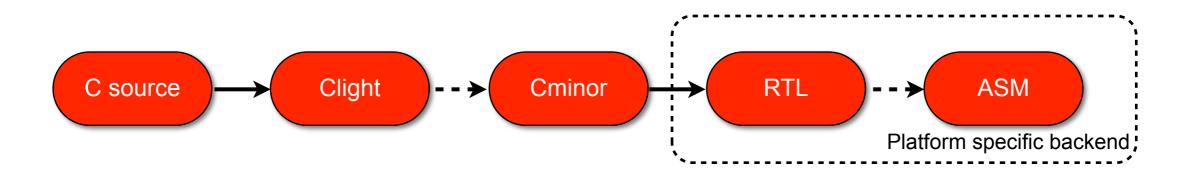
Why a value analysis for CompCert ?

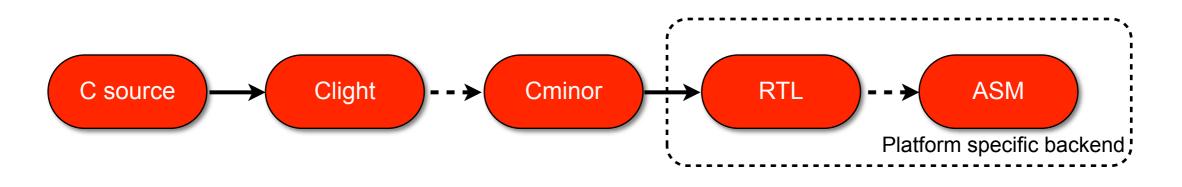
CompCert provides strong guarantees but only for programs with well behaved behaviors



A powerful and verified static analysis aims at proving that a program only exhibits well behaved behaviors

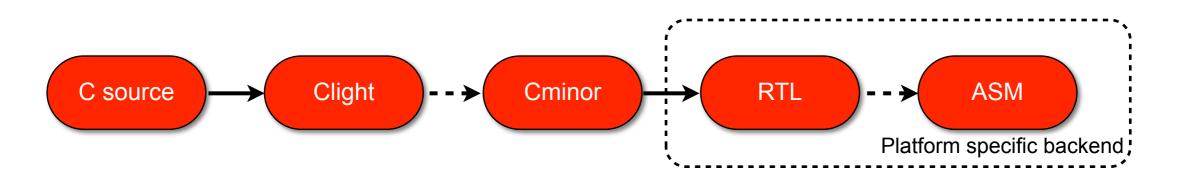
Analyser Theorem Theorem analyzer_is_correct: ∀ p, analyzer p = Success \rightarrow (\forall beh, exec_C_program p beh \rightarrow not_wrong beh).





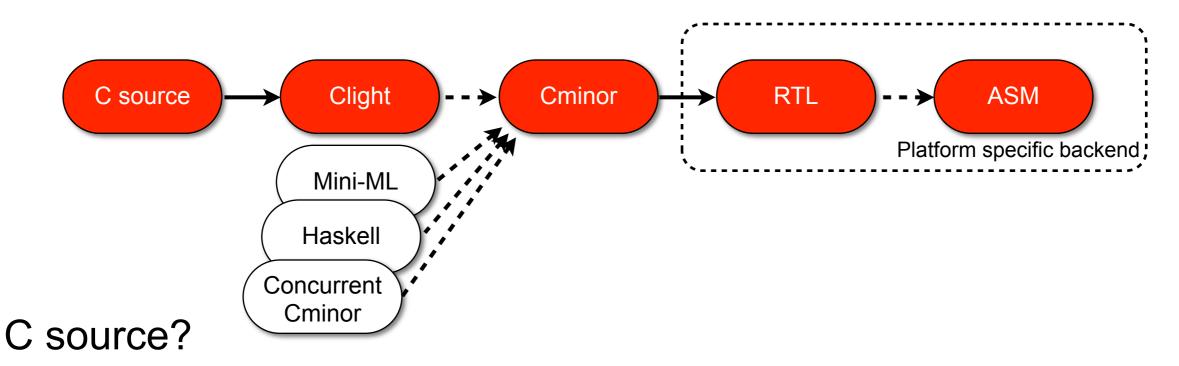
C source?

- the place where we want to prove program safety
- but the most difficult place to start (not an IR but a source language)



C source?

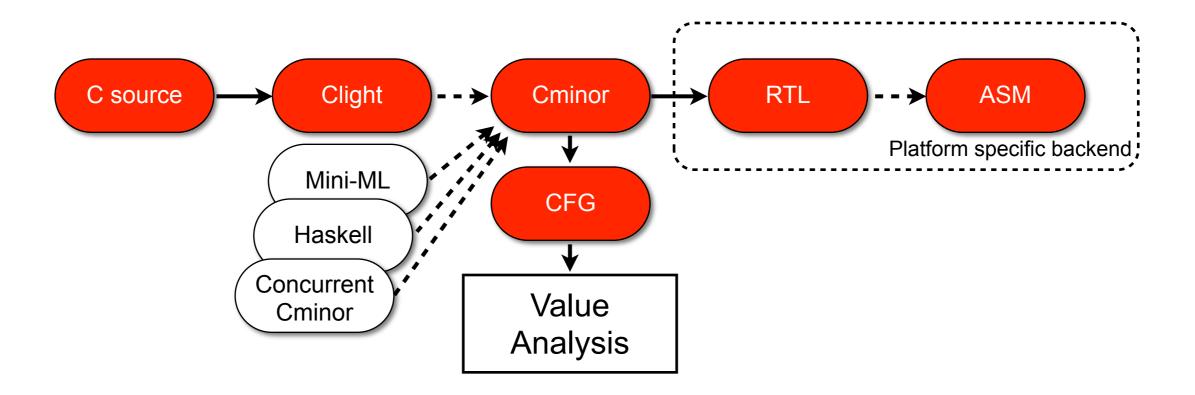
- the place where we want to prove program safety
- but the most difficult place to start (not an IR but a source language) RTL?
 - the place where most CompCert static analyses take place
 - but platform specific, flat expressions



- the place where we want to prove program safety
- but the most difficult place to start (not an IR but a source language) RTL?
 - the place where most CompCert static analyses take place
 - but platform specific, flat expressions

Cminor?

- the last step before platform specific semantics
- designed to welcome forthcoming extensions
- but control flow still less uniform than in RTL (nested blocks and exits)



CFG (Control Flow Graph)

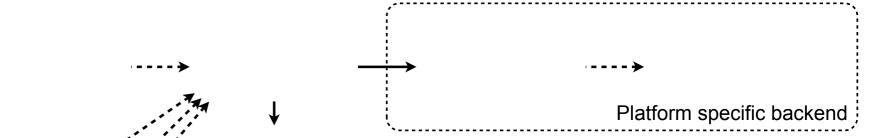
- a new representation recently added by JH. Jourdan and Xavier Leroy
- Cminor expressions (i.e., side-effect free C expressions)
- control flow graphs with explicit program points
- control flow is restricted to simple unconditional and conditional jumps
- platform independent

CFG syntax

Statements:

$$\begin{split} i &::= \texttt{skip}(l) \\ & | \texttt{assign}(id, a, l) \\ & | \texttt{store}(\kappa, a, a, l) \\ & | \texttt{if}(a, l_{true}, l_{false}) \\ & | \texttt{call}(sig, id^?, a, a*, l) \\ & | \texttt{return}(a)^? \end{split}$$

no operation (go to l) assignment memory store if statement function call function return



CFG syntax

Statements:

$$\begin{split} i &::= \texttt{skip}(l) \\ &| \texttt{assign}(id, a, l) \\ &| \texttt{store}(\kappa, a, a, l) \\ &| \texttt{if}(a, l_{true}, l_{false}) \\ &| \texttt{call}(sig, id^?, a, a*, l) \\ &| \texttt{return}(a)^? \end{split}$$

Expressions:

$$= id \\ | c \\ | op_1 a \\ | a_1 op_2 a_2 \\ | a_1? a_2: a_3 \\ | load(\kappa, a)$$

a ::

no operation (go to l) assignment memory store if statement function call function return

variable identifier constant unary arithmetic operation binary arithmetic operation conditional expression memory load

Platform specific backend

CFG syntax

Constants: $c ::= n \mid f$ $\mid \texttt{addrsymbol}(id,n)$ addrstack(n)Unary op.: $op_1 ::= cast8unsigned$ cast8signed cast16unsigned cast16signed boolval negint notbool notint Binary op.: $op_2 ::= + |-| * | / | %$ $| << | >> | \& | | | ^{}$ $| /_{u} | %_{u} | >>_{u}$ $| \operatorname{cmp}(b)$ $\mathtt{cmpu}(b)$ Comparisons: b ::= < | <= | > | >= | == | != relational operators

integer and floating-point constants address of a symbol plus an offset stack pointer plus a given offset

8-bit zero extension 8-bit sign extension 16-bit zero extension 16-bit sign extension 0 if null, 1 if non-null integer opposite boolean negation bitwise complement arithmetic integer operators bitwise operators unsigned operators integer signed comparisons integer unsigned comparisons

CFG semantics

A CFG program manipulate values

- Vint i with i an integer
- Vptr b i with a memory block identifier and i an offset (integer)
- Vfloat f with f a floating-point number
- Vundef (contents of uninitialized memory)

An integer is modelled with a dependent record

```
Definition wordsize: nat := 32.
Definition modulus : Z := two_power_nat wordsize.
Definition half_modulus : Z := modulus / 2.
Definition max_unsigned : Z := modulus - 1.
Definition max_signed : Z := half_modulus - 1.
Definition min_signed : Z := - half_modulus.
```

Record int := { intval: Z; intrange: 0 <= intval < modulus }.

and a signed and an unsigned interpretation.

CFG semantics

A small step semantics step: state -> trace -> state -> Prop models program execution (with a trace of behavior)

```
Inductive step: state \rightarrow trace \rightarrow state \rightarrow Prop :=
  | step_skip:
       \forall s f sp pc e m pc',
       (fn_code f) !pc = Some(Iskip pc') \rightarrow
       step (State s f sp pc e m)
         E0 (State s f sp pc' e m)
  | step_assign:
       \forall s f sp pc e m id a pc' v,
       (fn_code f)!pc = Some(Iassign id a pc') \rightarrow
       eval_expr ge sp e m a v 
ightarrow
       step (State s f sp pc e m)
         E0 (State s f sp pc' (PTree.set id v e) m)
  | step_store:
       \forall s f sp pc e m chunk addr src pc' vaddr vsrc m',
       (fn_code f)!pc = Some(Istore chunk addr src pc') \rightarrow
       eval_expr ge sp e m addr vaddr 
ightarrow
       eval\_expr ge sp e m src vsrc \rightarrow
       Mem.storev chunk m vaddr vsrc = Some m' 
ightarrow
       step (State s f sp pc e m)
         E0 (State s f sp pc' e m')
  [...]
```

CFG Analyser

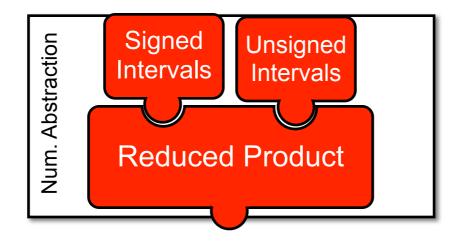
For all reachable states (according to step), for all all local variables that contain a value (Vint i) or (Vptr b i), the analyser computes a range for the integer i.

In the future, these inferred properties will be used to prove execution safety.

Abstractions currently handled

- interval abstraction for signed and unsigned interpretation of integers
- each local variable is abstracted by a pair of intervals.

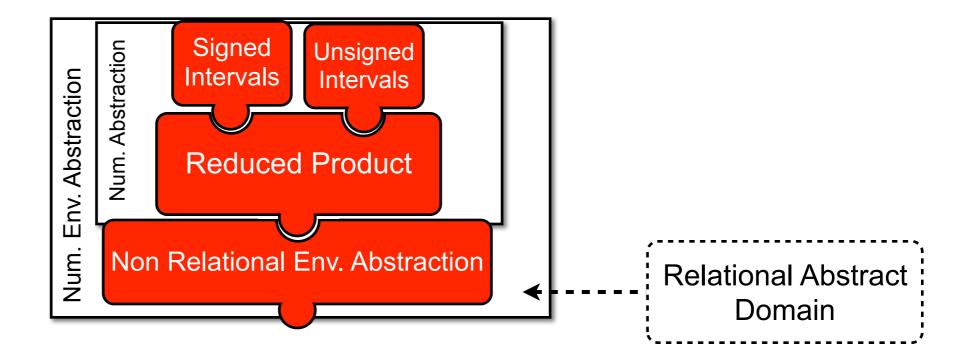
But the main contribution of the current work is a set of Coq interfaces



Numeric Abstraction

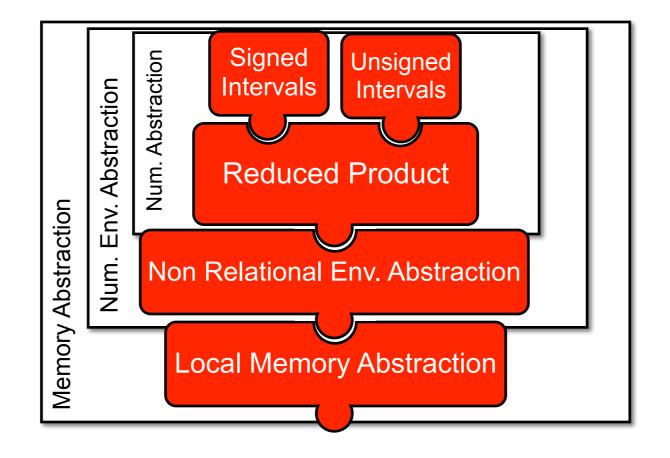
- abstraction of a single integer value,
- takes into account all the CompCert numerical operations
- reduced product: combine two abstractions for better precision Example:

if signed(i) $\in [-256, 255] \land \text{unsigned}(i) \in [512, 2^{32} - 1]$ then signed(i) $\in [-256, -1] \land \text{unsigned}(i) \in [2^{32} - 256, 2^{32} - 1]$



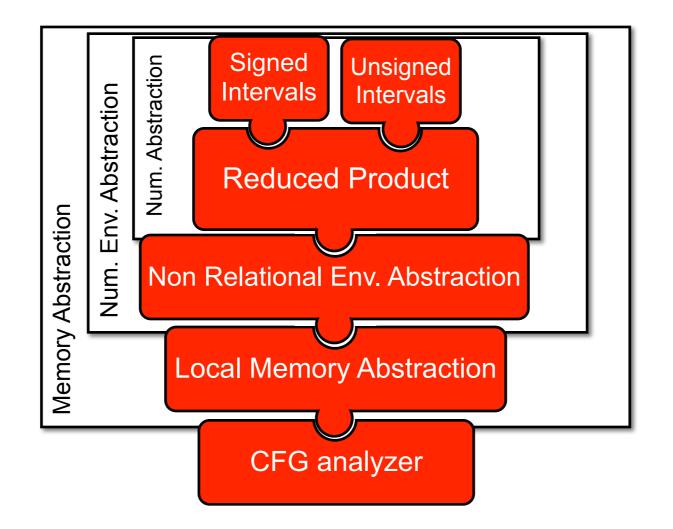
Numeric Environment Abstraction

- parameterised by an abstract notion of variable
- instantiated with a non-relational abstraction (each variable is given a numerical abstraction)
- interface ready for relational abstraction



Memory Abstraction

- the only signature where the C memory is exposed
- currently implemented by mapping V with local variables
- ready for more ambitious abstractions where V is also mapped to memory cells



Generic analyzer

- parameterised by any memory abstraction
- CFG program are unstructured: need to build widening strategies on unstructured control flow graph
 => we let an external tool computes a post-fixpoint and check the result in Coq

Conclusions

A first step toward a verified C verifier

 experimental evaluations show that our tool compare already well with Frama-C value analysis (more in the paper)

Next steps

- abstract interpretation a source level
- relational abstract domains
- abstract domains for floating-point numbers

Experiments on safety critical programs

- stress test the efficiency of the analyser
- add new abstract domains for specific program patterns

