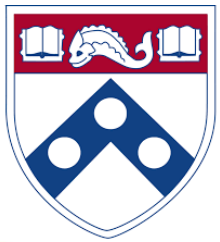
Vellvm: Verifying the LLVM IR

Steve Zdancewic
University *of* Pennsylvania

Dagstuhl Seminar on
Secure Compilation
May 2018

Collaborators

- Jianzhou Zhao
- Milo M.K. Martin
- Santosh Nagarakatte
- Dmitri Garbuzov
- William Mansky
- Christine Rizkallah
- Yannick Zakowski
- Olek Gierczak

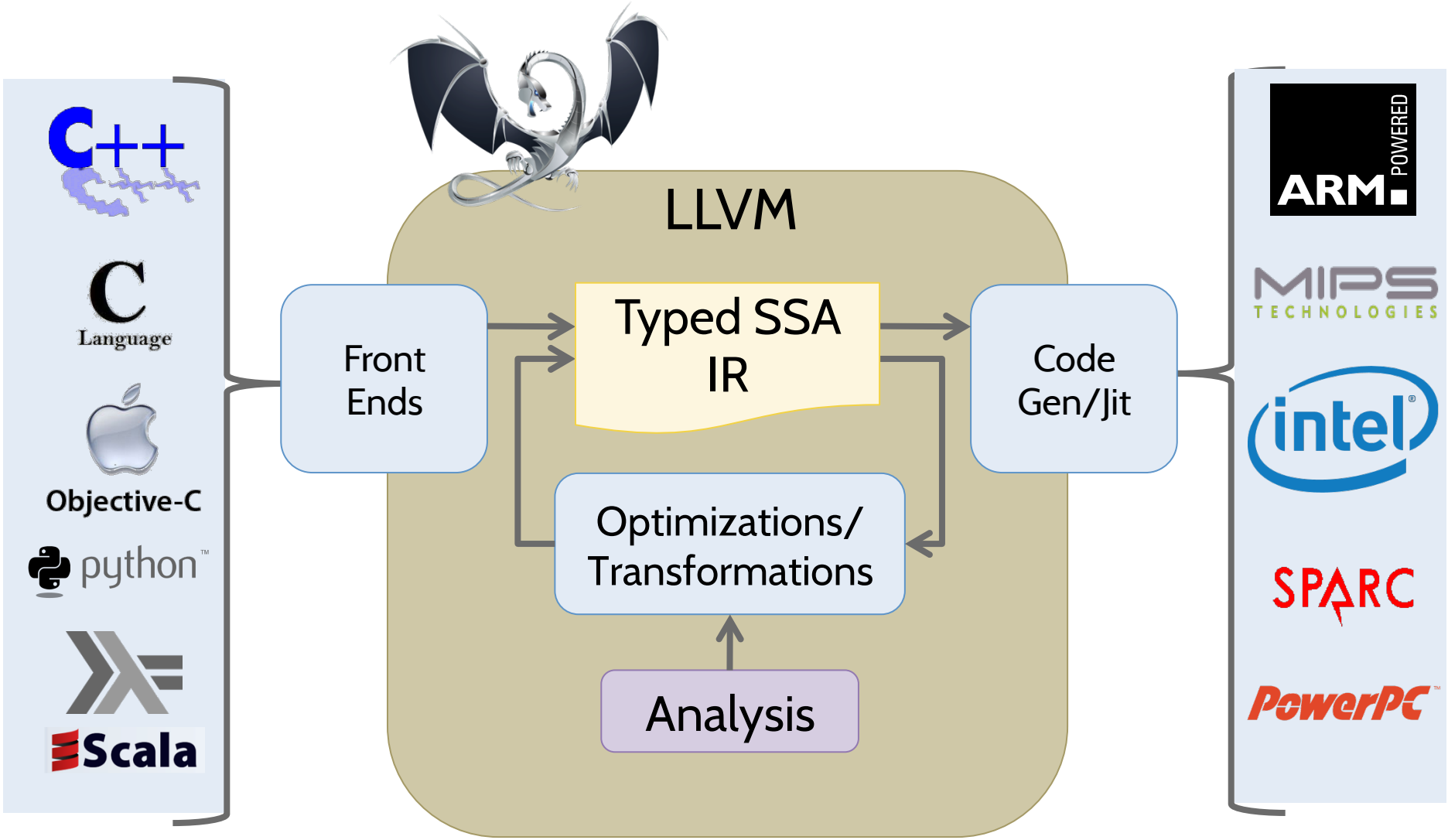


Google

- Gil Hur
- Jeehon Kang
- Viktor Vafeiadis

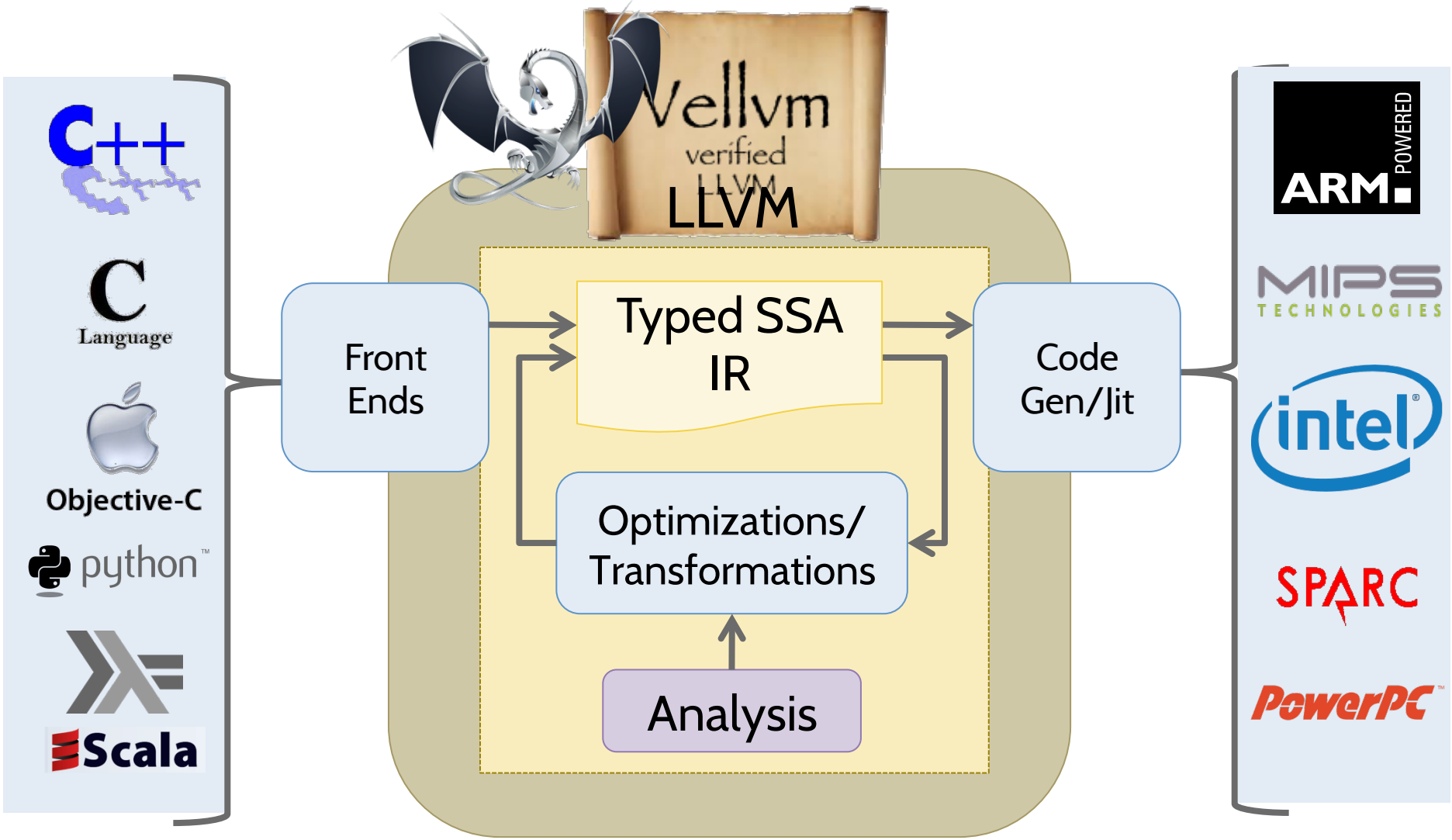
LLVM Compiler Infrastructure

[Lattner et al.]



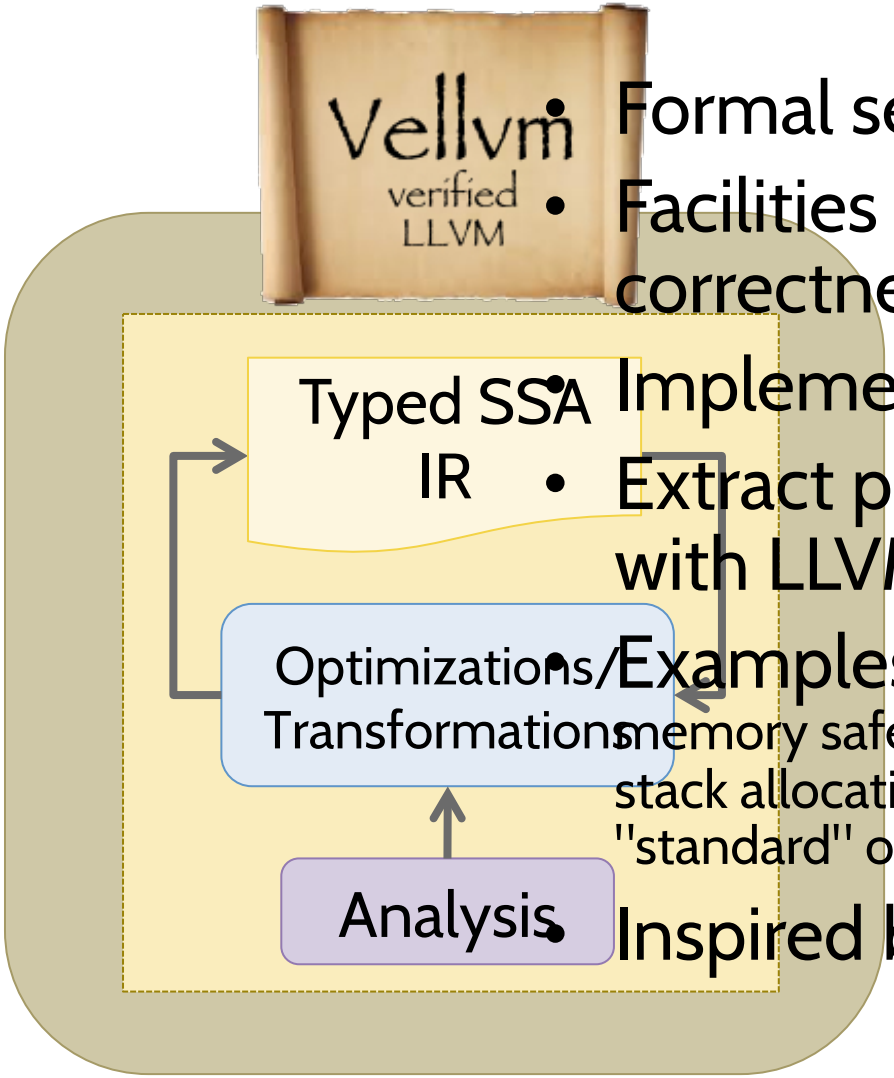
LLVM Compiler Infrastructure

[Lattner et al.]



The Vellvm Project

[Zhao et al. POPL 2012, CPP 2012, PLDI 2013]



- Formal semantics
- Facilities for creating correctness proofs
- Implemented in Coq
- Extract passes for use with LLVM compiler
- Examples:
  - memory safety
  - stack allocation elimination
  - "standard" optimizations
- Inspired by CompCert

Vellvm Framework

Coq

Type System and SSA

Operational Semantics

Syntax

Memory Model

Proof Techniques & Metatheory

Extract

OCaml

OCaml Bindings

Parser

Printer

LLVM

C Source Code

LLVM IR

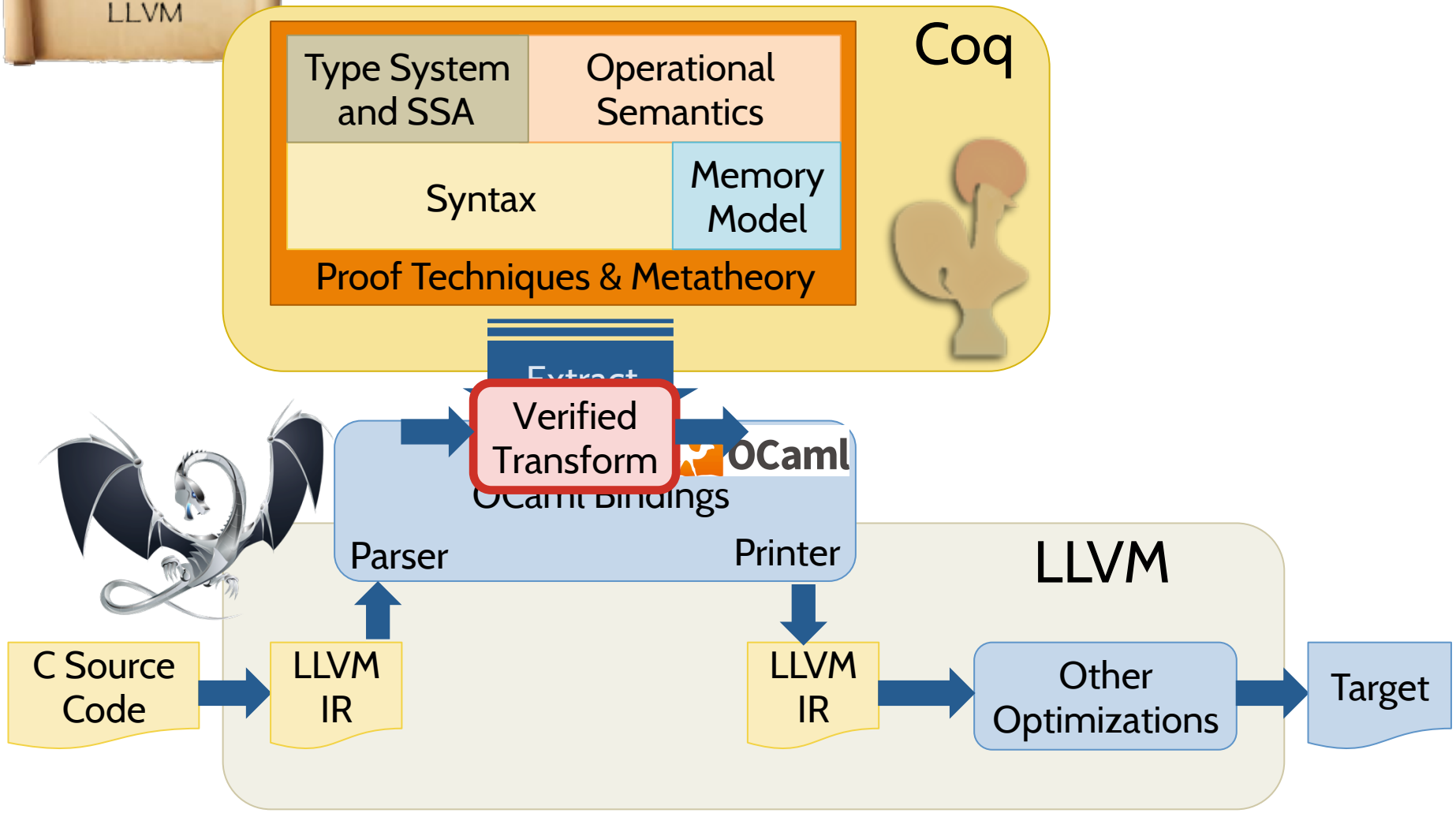
Transform

LLVM IR

Other Optimizations

Target

Vellvm Framework



Vellvm

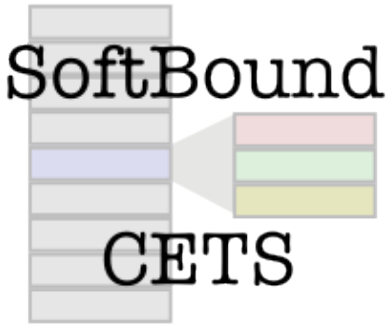
verified LLVM

- The Good
- The Bad
- The Ugly

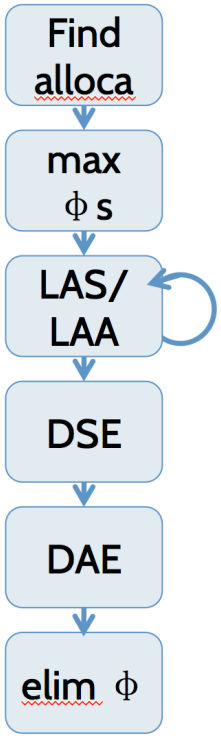
Research opportunities!

The Good

- Verified SoftBound [POPL 2012]
  - Memory Safety



- Verified mem2reg [PLDI 2013]
  - Register promotion, defined in terms of a stack of "micro-optimizations"



- Verified dominator analysis [CPP 2012]
  - Cooper-Harvey-Kennedy Algorithm [2000]

Operational Semantics

	Small Step	Big Step
Nondeterministic	LLVM_{ND}	
Deterministic	$\text{LLVM}_{Interp} \approx \text{LLVM}_{D} \gtrsim \text{LLVM}^{*}_{DFn} \gtrsim \text{LLVM}^{*}_{DB}$	

Proof-of-concept formal semantics.
Simulation up to "observable events."

The Bad

- Large, monolithic code base
- Clunky proofs

Development	LOC (Defns. + Proofs)
syntax + semantics	~45K
mem2reg + optimizations	~60K
SoftBound	~15K

⇒ hard for others (and us!) to adopt/adapt

The Bad (2)

- Representation & semantics
  - "proof of concept" not well engineered
- Limited use of proof automation
- Coq features (e.g. typeclasses) not available

⇒ Hard to deal with change:

LLVM 2.6 ⇒ LLVM 3.0 ⇒ LLVM 3.6 ⇒ ...

Coq 8.2 ⇒ Coq 8.3 ⇒ Coq 8.4 ⇒ Coq 8.5 ⇒...

CompCert's memory model evolution

The ugly

Target-dependent Results

- Undefined values:

```
%v = add i32 %x, undef
```

- Uninitialized memory:

```
%ptr = alloca i32
%v = load (i32*) %ptr
```

- Ill-typed memory usage / ptr2int / int2ptr

Nondeterminism

Fatal Errors

- Out-of-bounds accesses
- Access dangling pointers
- Free invalid pointers
- Invalid indirect calls

Stuck States

The Ugly

Target-dependent Results

- Undefined values:

```
%v = add i32 %x, undef
```

- Uninitialized memory:

```
%ptr = alloca i32
%v = load (i32*) %ptr
```

- Ill-typed memory usage / ptr2int / int2ptr

Nondeterminism

Thorny semantic issues.

Defined by a predicate on the program configuration.

Stuck(f, σ) = BadFree(f, σ)
˅ BadLoad(f, σ)
˅ BadStore(f, σ)
˅ ...
˅ ...

Stuck States

The Ugly (2)

- Undef semantics may change for the better!
  - existing transformations are unsound
  - Gil & others: "freeze"
- Relaxed consistency memory models
  - at the LLVM IR level
- LLVM "feature creep"
  - focus on performance
  - many annotations to accommodate needs of different sources/targets
- LLVM is a quickly moving target!

⇒ Not feasible to cover all of modern LLVM (at least initially, maybe ever)

Vellvm II

- Re-engineered Vellvm development
  - Cleaner design
  - More modular
  - Extractable executable interpreter
  - Semantic representation compatible with other DeepSpec Projects
  - Aim to cover a large(-ish), reasonable subset of LLVM IR
- Experimental activities:
  - Change of representation to support higher-level reasoning principles

https://github.com/vellvm/vellvm

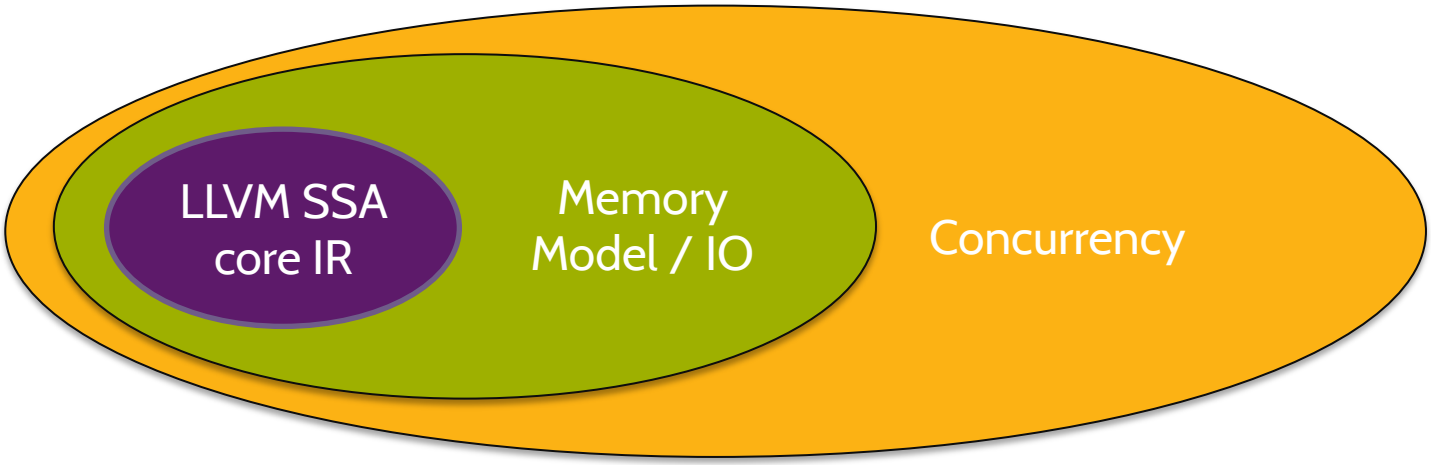
Plan

1. Tour of Vellvm's new structure

2. Experiments with with Call-By-Push-Value

Modular Semantics

- Factor out memory model [CAV 15]
  - similar to linking/separate compilation
- For:
  - concurrency
  - more extensibility/robustness to changes
  - better support for casts [PLDI 15]



TopLevel.v

```
Require Import Vellvm.Classes.
Require Import Vellvm.LLVMIO.
Require Import Vellvm.StepSemantics.
Require Import Vellvm.Memory.

Module IO := LLVMIO.Make(Memory.A).
Export IO.DV.

Module M := Memory.Make(IO).
Module SS := StepSemantics(Memory.A)(IO).

Definition run_with_memory prog : option (IO.Trace IO.DV.dvalue) :=
  let scfg := Vellvm.AstLib.modul_of_toplevel_entities prog in
  match CFG.mcfg_of_modul scfg with
  | None ⇒ None
  | Some mcfg ⇒
    mret
      (M.memD M.empty
      ('s ← SS.init_state mcfg "main";   (* TODO: add argv, argc *)
        SS.step_sem mcfg (SS.Step s)))
  end.
```

LLVMIO.v

```
Inductive IO : Type → Type :=
| Alloca : ∀ (t:dtyp), (IO dvalue)
| Load   : ∀ (t:dtyp) (a:dvalue), (IO dvalue)
| Store  : ∀ (a:dvalue) (v:dvalue), (IO unit)
| GEP    : ∀ (t:dtyp) (v:dvalue) (vs:list dvalue), (IO dvalue)
| ItoP   : ∀ (i:dvalue), (IO dvalue)
| PtoI   : ∀ (a:dvalue), (IO dvalue)
| Call   : ∀ (t:dtyp) (f:string) (args:list dvalue), (IO dvalue)
.

(* Trace of events generated by a computation. *)
Definition Trace X := M IO X.
```

```
CoInductive M (Event : Type -> Type) X :=
| Ret (x:X)
| Vis {Y: Type} (e : Event Y) (k : Y -> M Event X)
| Tau (k: M Event X)
| Err (s:String.string)
.
```

The "freer" monad [Kiselyov, Ishii]

StepSemantics.v

```
Definition step (s:state) : Trace result :=
  let '(g, pc, e, k) := s in
  let eval_exp top exp := eval_exp g e top exp in

  do cmd ← trywith ("CFG has no instruction at " ++ string_of pc) (fetch CFG pc);
  match cmd with
  | Term (TERM_Ret (t, op)) ⇒ ...

  | Inst instruction ⇒
    do pc_next ← trywith "no fallthrough instruction" (incr_pc CFG pc);
    match (pt pc), instruction  with

      | IId id, INSTR_Op op ⇒
        'dv ← eval_op g e op;
         cont (g, pc_next, add_env id dv e, k)

      | IId id, INSTR_Alloca t _ _ ⇒
        Trace.Vis (Alloca (eval_typ t))
                  (λ dv ⇒ cont (g, pc_next, add_env id a e, k))

      | IId id, INSTR_Load _ t (u,ptr) _ ⇒
        'dv ← eval_exp (Some (eval_typ u)) ptr;
         Trace.Vis (Load (eval_typ t) dv)
```

External calls into the memory model.

Memory.v

```
CoFixpoint memD {X} (m:memory) (d:Trace X) : Trace X :=
  match d with
  | Trace.Tau d' ⇒ Trace.Tau (memD m d')
  | Trace.Vis _ io k ⇒
    match mem_step io m with
    | inr (m', v) ⇒ Trace.Tau (memD m' (k v))
    | inl e ⇒ Trace.Vis io k
    end
  | Trace.Ret x ⇒ d
  | Trace.Err x ⇒ d
  end.
```


```
Definition mem_step {X} (e:IO X) (m:memory) : (IO X) + (memory * X) :=
  match e with
  | Alloca t ⇒
    let new_block := make_empty_block t in
    inr  (add (size m) new_block m,
          DVALUE_Addr (size m, 0))

  | …
```

Many,Many Things Still TODO

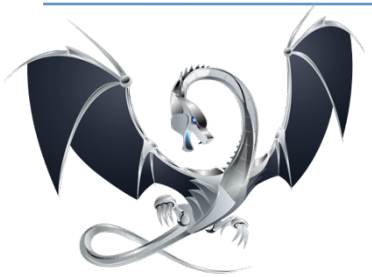
- LLVM Features
  - undef
  - richer, more accurate memory models
  - concurrency
  - switch / invoke & unwind
- Validation of Semantics
  - more testing / test cases
- Proofs…
  - about IR transformations
  - using Vellvm as target language

Plan

1. Tour of Vellvm's new structure

2. Experiments with with Call-By-Push-Value

LLVM IR Semantics



SSA ≈ functional program [Appel 1998]

+

- Effects
  - structured heap load/store
  - system calls (I/O)
- Types & Memory Layout
  - structured, recursive types
  - type-directed projection
  - type casts

We know (more or less) how to model this and prove properties about the models.

Operational Semantics for LL-IRs

$$G \vdash \langle p, e, k \rangle \longrightarrow \langle p', e', k' \rangle$$

- G is the program (e.g. a control-flow graph)
- p is the program counter
- e is the local environment (e.g. registers)
- k is the control stack (e.g. stack of call frames)

Note: global memory (& other effects) omitted for simplicity.

Example Rule: CALL

$$
G[p] = \begin{array}{l} o:\ a = \ldots \\ p:\ x = \text{CALL}\ \ f\ \ a \\ r:\ \ldots \end{array}
$$

$eval(f, e) = [p_f, e_f]$

$eval(a, e) = v$

$$G \vdash \langle p, e, k \rangle \longrightarrow \langle p_f, e_f,\ \ v::[r,x,e]::k \rangle$$

Function Inlining

$G[p_f]$ =	p_f: x = POP	$G'[p_f]$ =	p_f: x = POP	$G''[p_f]$ =	p_f: x = POP
	q_f: y = ADD x x		q_f: y = ADD x x		q_f: y = ADD x x
	r_f: RET y		r_f: RET y		r_f: RET y
	o: a = ...		o: a = ...		o: a = ...
G[p] =	p: x = CALL f a	$G'[p_0]$ =	p_0: PUSH a		~~p_0: PUSH a~~
	r: ...x...		p_1: x' = POP		~~p_1: x' = POP~~
			p_2: y' = ADD x' x'	$G''[p_2]$ =	p_2: y' = ADD a a
			p_3: PUSH y'		~~p_3: PUSH y'~~
		G'[p] =	p: x = POP		~~p: x = POP~~
			r: ...x...		r: ...y'...

known call

Inline the function body: rename variables & introduce extra PUSH/POP instructions to patch up the call interface

Remove PUSH/POP pairs, substitute defs for uses.

Proving This Correct?

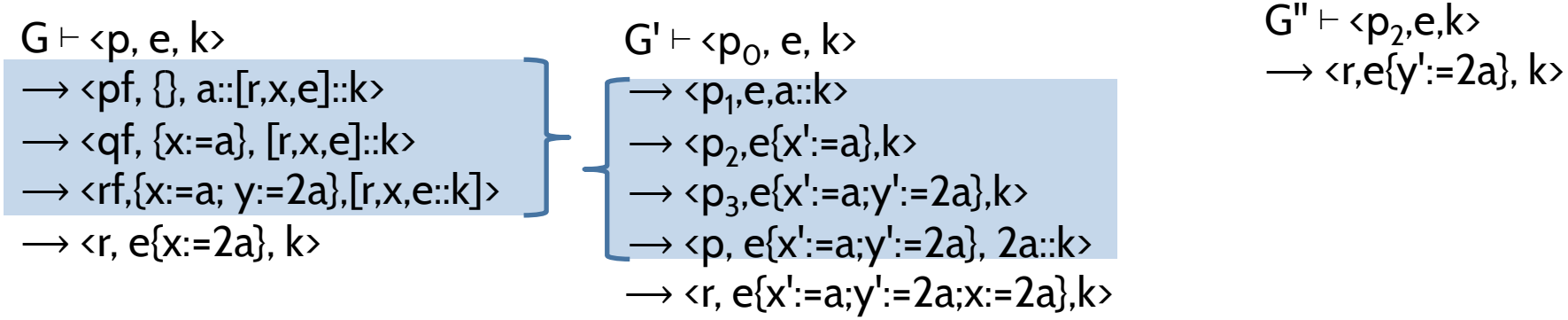
$G[p_f]$ =	p_f: x = POP	$G'[p_f]$ =	p_f: x = POP	$G''[p_f]$ =	p_f: x = POP
	q_f: y = ADD x x		q_f: y = ADD x x		q_f: y = ADD x x
	r_f: RET y		r_f: RET y		r_f: RET y
	o: a = ...		o: a = ...		o: a = ...
G[p] =	p: x = CALL f a	$G'[p_0]$ =	p_0: PUSH a		~~p_0: PUSH a~~
	r: ...x...		p_1: x' = POP		~~p_1: x' = POP~~
			p_2: y' = ADD x' x'	$G''[p_2]$ =	p_2: y' = ADD a a
			p_3: PUSH y'		~~p_3: PUSH y'~~
		G'[p] =	p: x = POP		~~p: x = POP~~
			r: ...x...		r: ...y'...

$G \vdash \langle p, e, k\rangle \longrightarrow \langle pf, \{\}, a::[r,x,e]::k\rangle \longrightarrow \langle qf, \{x:=a\}, [r,x,e]::k\rangle \longrightarrow \langle rf, \{x:=a;\ y:=2a\}, [r,x,e::k]\rangle$
$\longrightarrow \langle r, e\{x:=2a\}, k\rangle$

$G' \vdash \langle p_0, e, k\rangle \longrightarrow \langle p_1, e, a::k\rangle \longrightarrow \langle p_2, e\{x':=a\}, k\rangle \longrightarrow \langle p_3, e\{x':=a;y':=2a\}, k\rangle$
$\longrightarrow \langle p, e\{x':=a;y':=2a\}, 2a::k\rangle \longrightarrow \langle r, e\{x':=a;y':=2a;x:=2a\}, k\rangle$

$G'' \vdash \langle p_2, e, k\rangle \longrightarrow \langle r, e\{y':=2a\}, k\rangle$

Building a Simulation



- Nontrivial mapping of pc's in G to pc's in G' (and G' and G'')
  - inside of f and outside of f
- Uses freshness
  - x', y' not used in the continuation of G' starting at r
  - to relate the *environments* of G and G'
  - to relate the *code* of G, G', and G''
- Needs *renaming* substitutions: G''[r] ~ G[r]{y'/x}

Upshot: Nontrivial proofs, lots of bookkeeping,
not very compositional.

Can we do better?

Lambda Calculus + Equational Theory

$$\texttt{let f = (fun x -> x + x) in C[f a]}$$

$$\cong$$

$$\texttt{let f = (fun x -> x + x) in C[(fun x -> x+x) a]}$$

$$\cong$$

$$\texttt{let f = (fun x -> x + x) in C[a + a]}$$

$\cong$ is contextual equivalence:

beta:
$(\texttt{fun x -> e})\ v \cong e\{v/x\}$

congruence:
$e_1 \cong e_2 \Rightarrow C[e_1] \cong C[e_2]$

identity:
$e \cong e$

Call-By-Push-Value

[Levy 1999]

- Simple, structural operational semantics
- Equational reasoning
  - contextual equivalence
  - substitution
- Extensible in the "usual" ways
  - algebraic datatypes
  - effects (state/control)
- "Low" Level Lambda Calculus
  - distinguishes between code labels and closures
  - direct / indirect jump
  - a bit like CPS (but retains the stack)
- Good connections to type theory
  - program analysis as type systems?

OCaml

```
let rec
  mult n x a =
    if x = 0 then 0 else
    let y = x - 1 in
      if y = 0 then a else
      let b = a + n in
      mult n y b
in
  mult
```

CFG

```
0:   RET @1

1:   n = POP [2]
2:   x = POP [3]
3:   a = POP [4]
4:   CBR x [8] [5]

5:   y = SUB x 1 [6]
6:   CBR y [7] [9]
7:   RET a

8:   RET 0

9:   b = ADD a n [10]
10:  TAIL @1 b y n
```

Call-by-Push-Value

letrec
$$mult = \lambda n.\lambda x.\lambda a.$$
if0 x (**prd** 0)
$(x - 1$ **to** y **in**
(**if0** y (**prd** a)
$(a + n$ **to** b **in**
$n \cdot y \cdot b \cdot mult)))$
in
$mult$

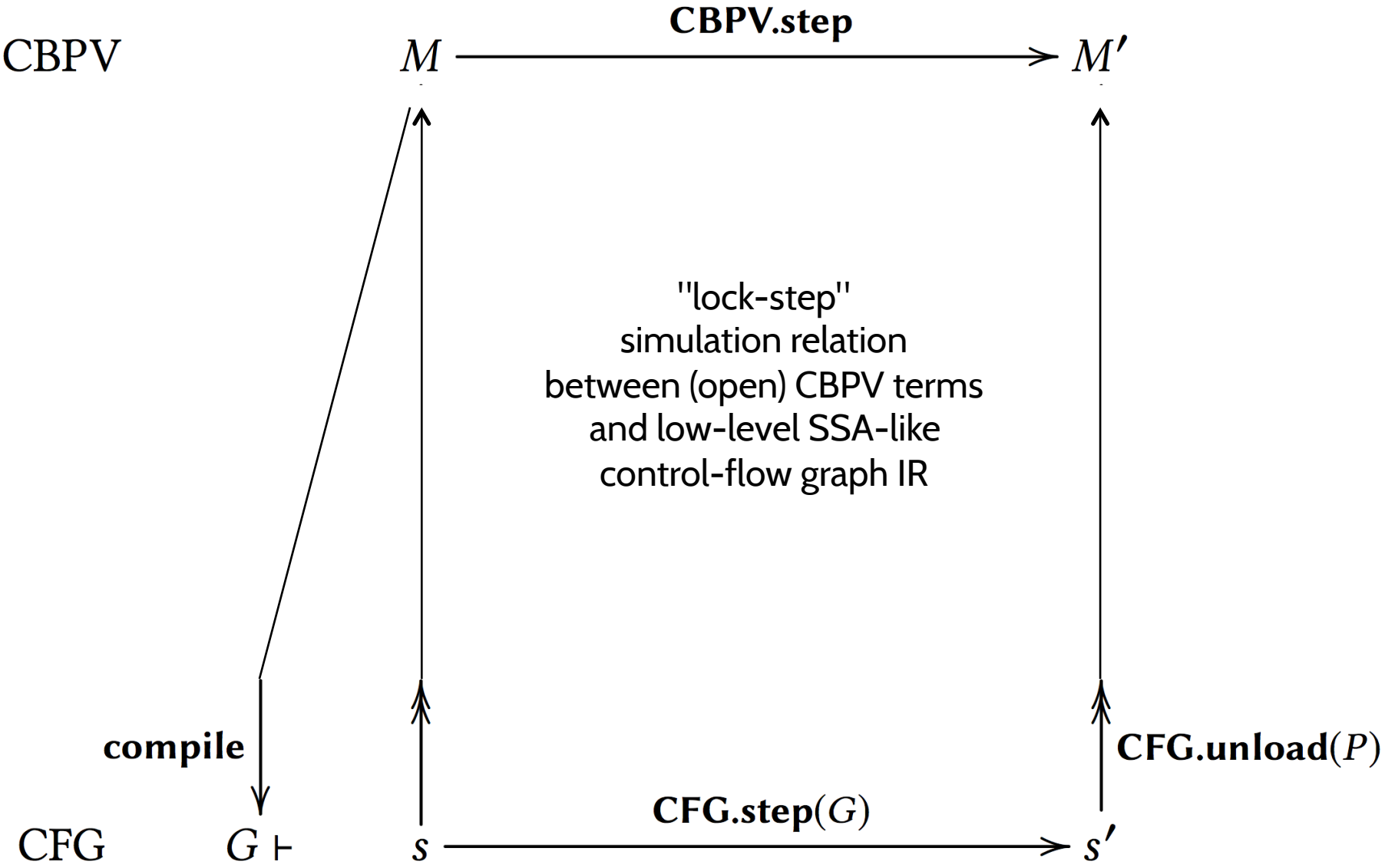
$$
\begin{array}{lllll}
\text{Values} \ni V & ::= & x \mid n & \mid & \mathbf{thunk}\, M \\
\text{Terms} \ni M, N & ::= & \mathbf{force}\, V & \mid & \mathbf{letrec}\, x_1 = M_1, \ldots, x_n = M_n\, \mathbf{in}\, N \\
& \mid & \mathbf{prd}\, V & \mid & M\, \mathbf{to}\, x\, \mathbf{in}\, N \\
& \mid & V \cdot M & \mid & \lambda x.M \\
& \mid & V_1 \oplus V_2 & \mid & \mathbf{if0}\ V\ M_1\ M_2
\end{array}
$$

Syntactic distinction between *value* and *computation* terms.

Monadic structure: $\mathbf{prd}\ V$ is the "return"
$M\ \mathbf{to}\ x\ \mathbf{in}\ N$ is the "bind"

+Mutually recursive computations

Machine Correspondence Theorem



If $e_1 \cong e_2$ then (compile e_1) $\cong$ (compile e_2).

contextual equivalence

appropriate bisimilarity

Strong Correspondence:

Any notion of "observation" of the source program is preserved by compilation.

CBPV Equation	CFG Optimization
$\mathbf{force}\,(\mathbf{thunk}\,M) = M$	block merging “direct jump case"
$V \cdot \lambda x.M = \{V/x\}\,M$	block merging “phi case"
$\mathbf{prd}\,V\,\mathbf{to}\,x\,\mathbf{in}\,M = \{V/x\}\,M$	move elimination
$(n_1 \oplus n_2)\,\mathbf{to}\,x\,\mathbf{in}\,M = \{n_1 [\![\oplus]\!] n_2/x\}\,M$	constant folding
$\mathbf{thunk}\,(\lambda y.M) \cdot \lambda x.N = \{\mathbf{thunk}\,(\lambda y.M)/x\}\,N$	function inlining
$\mathbf{if0}\ 0\ M_1\ M_2 \longrightarrow M_1$	dead branch elimination “true branch"
$\mathbf{if0}\ n\ M_1\ M_2 \longrightarrow M_2$ where $(n \neq 0)$	dead branch elimination “false branch"
$\mathbf{if0}\ V\ M\ M \longrightarrow M$	branch elimination

Many CFG transformations can be proven correct
by reasoning in the equational theory of CBPV.

Work in Progress

- Back-translation of (subset of) Vellvm IR to CBPV
  - "decompile" LLVM into CBPV
  - Aim: (compile(decompile(e)) ≅ e
- Motivates some tweaks to CBPV level
  - CBPV to LLVM IR compiler independently interesting?

Vellvm

https://github.com/vellvm/vellvm

deep spec

deepspec.org

PENN PL