Provably Safe Systems

Prospects and Approaches

Mario Carneiro

Chalmers University of Technology

September 4, 2024

Disclaimers

- I am an AI outsider
- I am an ITP power user
- This presentation is mostly speculative and opinion-based
  - ...but I did find some opportunity to hawk my wares in here
- This is not my area of expertise
  - Shoutout to Adam Vandervorst and Anneline Daggelinckx, who have also written on this topic and are probably better informed than me about it

Provably safe systems

PROVABLY SAFE SYSTEMS: THE ONLY PATH TO CONTROLLABLE AGI

Max Tegmark
Department of Physics
Insitute for AI & Fundamental Interactions
Massachusetts Institute of Technology
Cambridge, MA 02139

Steve Omohundro
Beneficial AI Research
Palo Alto, CA 94301

September 6, 2023

ABSTRACT

We describe a path to humanity safely thriving with powerful Artificial General Intelligences (AGIs) by building them to provably satisfy human-specified requirements. We argue that this will soon be technically feasible using advanced AI for formal verification and mechanistic interpretability. We further argue that it is the only path which guarantees safe controlled AGI. We end with a list of challenge problems whose solution would contribute to this positive outcome and invite readers to join in this work.

https://arxiv.org/abs/2309.01933

Provably safe systems

- Josef suggested I give a review / response of this paper for the AITP crowd. Let's see how it goes

Provably safe systems

- Josef suggested I give a review / response of this paper for the AITP crowd. Let's see how it goes

Summary

Provably safe systems

- Josef suggested I give a review / response of this paper for the AITP crowd. Let's see how it goes

Summary

- Misaligned AGI is an existential threat

Provably safe systems

- Josef suggested I give a review / response of this paper for the AITP crowd. Let's see how it goes

Summary

- Misaligned AGI is an existential threat
- Properly addressing this threat requires a "security mindset", treating AGIs as full adversaries

Provably safe systems

- Josef suggested I give a review / response of this paper for the AITP crowd. Let's see how it goes

Summary

- Misaligned AGI is an existential threat
- Properly addressing this threat requires a "security mindset", treating AGIs as full adversaries
- The solution is to use mathematical proof to ensure that deployed programs meet their specifications

Provably safe systems

- Josef suggested I give a review / response of this paper for the AITP crowd. Let's see how it goes

Summary

- Misaligned AGI is an existential threat
- Properly addressing this threat requires a "security mindset", treating AGIs as full adversaries
- The solution is to use mathematical proof to ensure that deployed programs meet their specifications
- The proofs can be constructed by AIs

Provably safe systems

- Josef suggested I give a review / response of this paper for the AITP crowd. Let's see how it goes

Summary

- Misaligned AGI is an existential threat
- Properly addressing this threat requires a "security mindset", treating AGIs as full adversaries
- The solution is to use mathematical proof to ensure that deployed programs meet their specifications
- The proofs can be constructed by AIs
- The algorithms satisfying those proofs are also constructed by AI

Provably safe systems

- Josef suggested I give a review / response of this paper for the AITP crowd. Let's see how it goes

Summary

- Misaligned AGI is an existential threat
- Properly addressing this threat requires a "security mindset", treating AGIs as full adversaries
- The solution is to use mathematical proof to ensure that deployed programs meet their specifications
- The proofs can be constructed by AIs
- The algorithms satisfying those proofs are also constructed by AI
  - In particular, the ML algorithms themselves are not the ones doing the job

Provably safe systems

- Josef suggested I give a review / response of this paper for the AITP crowd. Let's see how it goes

Summary

- Misaligned AGI is an existential threat
- Properly addressing this threat requires a "security mindset", treating AGIs as full adversaries
- The solution is to use mathematical proof to ensure that deployed programs meet their specifications
- The proofs can be constructed by AIs
- The algorithms satisfying those proofs are also constructed by AI
  - In particular, the ML algorithms themselves are not the ones doing the job
- Regulations should require that only verified components are deployed

My take on it

- I am significantly less hopeful than Tegmark and Omohundro that AIs will solve our problems
- But I agree with most of the main points:
  - Misaligned AGI is an existential threat

My take on it

- I am significantly less hopeful than Tegmark and Omohundro that AIs will solve our problems
- But I agree with most of the main points:
  - Misaligned AGI is an existential threat
  - Properly addressing this threat requires a “security mindset”, treating AGIs as full adversaries

My take on it

- I am significantly less hopeful than Tegmark and Omohundro that AIs will solve our problems
- But I agree with most of the main points:
  - Misaligned AGI is an existential threat
  - Properly addressing this threat requires a “security mindset”, treating AGIs as full adversaries
  - The solution is to use mathematical proof to ensure that deployed programs meet their specifications

My take on it

- I am significantly less hopeful than Tegmark and Omohundro that AIs will solve our problems
- But I agree with most of the main points:
  - Misaligned AGI is an existential threat
  - Properly addressing this threat requires a “security mindset”, treating AGIs as full adversaries
  - The solution is to use mathematical proof to ensure that deployed programs meet their specifications
  - The proofs can be constructed by ~~AIs~~ computer-assisted humans

My take on it

- I am significantly less hopeful than Tegmark and Omohundro that AIs will solve our problems
- But I agree with most of the main points:
  - Misaligned AGI is an existential threat
  - Properly addressing this threat requires a "security mindset", treating AGIs as full adversaries
  - The solution is to use mathematical proof to ensure that deployed programs meet their specifications
  - The proofs can be constructed by ~~AIs~~ computer-assisted humans
  - The algorithms satisfying those proofs are constructed by ~~AI~~ programmers
    - In particular, the ML algorithms themselves are not the ones doing the job

My take on it

- I am significantly less hopeful than Tegmark and Omohundro that AIs will solve our problems
- But I agree with most of the main points:
  - Misaligned AGI is an existential threat
  - Properly addressing this threat requires a "security mindset", treating AGIs as full adversaries
  - The solution is to use mathematical proof to ensure that deployed programs meet their specifications
  - The proofs can be constructed by ~~AIs~~ computer-assisted humans
  - The algorithms satisfying those proofs are constructed by ~~AI~~ programmers
    - In particular, the ML algorithms themselves are not the ones doing the job
  - Regulations should require that only verified components are deployed (?)

Proof-carrying code

- Tegmark and Omohundro envision a whole stack of proved components:
  - Proof-carrying code (PCC) is code which carries within it a proof of correctness
  - Provably Compliant Hardware (PCH)
  - "Provable Contracts (PC) govern physical actions by using secure hardware to provably check compliance with a formal specification before actions are taken"
  - "Provable Meta-Contracts (PMC) impose formal constraints on the creation or modification of other provable contracts"

Proof-carrying code

- Tegmark and Omohundro envision a whole stack of proved components:
  - Proof-carrying code (PCC) is code which carries within it a proof of correctness
  - Provably Compliant Hardware (PCH)
  - "Provable Contracts (PC) govern physical actions by using secure hardware to provably check compliance with a formal specification before actions are taken"
  - "Provable Meta-Contracts (PMC) impose formal constraints on the creation or modification of other provable contracts"
- This sounds great to me, but also slightly unrealistic
- There is not much care taken to restrict to areas where formal specification is feasible
  - Examples given in the paper stray very close to things like ethical principles that have been the subject of philosophical debate for centuries
  - It is certainly possible to have formal specifications for code and hardware, but this is generally limited to areas where design is "deliberate"

Proof-carrying code

- PCC itself (deploying the proof with the code) is not really needed here
- It also adds some self-referentiality to the proof: should the proof generation capacity of the code be part of the proof as well?
- To me, saying AI should write the proofs is overestimating the strength of AI and underestimating the capabilities of humans empowered with the right language design

Provably compliant hardware

- This is an area I would love to break into, but the lack of open source hardware makes things hard for people not at Intel et al.

Provably compliant hardware

- This is an area I would love to break into, but the lack of open source hardware makes things hard for people not at Intel et al.
- Key question: proved *to whom*?

Provably compliant hardware

- This is an area I would love to break into, but the lack of open source hardware makes things hard for people not at Intel et al.
- Key question: proved *to whom*?
- Zero-knowledge proofs to the rescue?

Provably compliant hardware

- This is an area I would love to break into, but the lack of open source hardware makes things hard for people not at Intel et al.
- Key question: proved *to whom*?
- Zero-knowledge proofs to the rescue?
- From what I know, this is already being done to some extent at Intel et al, because mistakes are extremely expensive

The troubles with proving stuff correct

- Specifications are harder and proofs are easier than the paper gives credit for

The troubles with proving stuff correct

- Specifications are harder and proofs are easier than the paper gives credit for
- The most common issue is that the proof cannot be completed because the system is in fact broken

The troubles with proving stuff correct

- Specifications are harder and proofs are easier than the paper gives credit for
- The most common issue is that the proof cannot be completed because the system is in fact broken
  - Row-Hammer is given as an example of a side channel to get around PCC

The troubles with proving stuff correct

- Specifications are harder and proofs are easier than the paper gives credit for
- The most common issue is that the proof cannot be completed because the system is in fact broken
  - Row-Hammer is given as an example of a side channel to get around PCC
  - ...but it is a bug in the hardware

The troubles with proving stuff correct

- Specifications are harder and proofs are easier than the paper gives credit for
- The most common issue is that the proof cannot be completed because the system is in fact broken
  - Row-Hammer is given as an example of a side channel to get around PCC
  - ...but it is a bug in the hardware
  - It's not a hardware spec bug, because the bad behavior should clearly not be user-visible

The troubles with proving stuff correct

- Specifications are harder and proofs are easier than the paper gives credit for
- The most common issue is that the proof cannot be completed because the system is in fact broken
  - Row-Hammer is given as an example of a side channel to get around PCC
  - ...but it is a bug in the hardware
  - It’s not a hardware spec bug, because the bad behavior should clearly not be user-visible
  - But the bug in physical implementation is not easily fixed

The troubles with proving stuff correct

- Specifications are harder and proofs are easier than the paper gives credit for
- The most common issue is that the proof cannot be completed because the system is in fact broken
  - Row-Hammer is given as an example of a side channel to get around PCC
  - ...but it is a bug in the hardware
  - It's not a hardware spec bug, because the bad behavior should clearly not be user-visible
  - But the bug in physical implementation is not easily fixed
- If you take this kind of project seriously, you quickly find that bugs exist all through the tech stack and the theorem you want to have is just not true. How to proceed?

Mechanistic interpretability (MI)

- Tegmark suggests that algorithms can be distilled from neural networks by a kind of computational neuroscience

Mechanistic interpretability (MI)

- Tegmark suggests that algorithms can be distilled from neural networks by a kind of computational neuroscience
- "Once a good algorithm has been discovered, neural networks lose their computational advantage. Indeed, PCCs may be more efficient, because neural networks have been shown to underperform GOFAI methods on basic tasks such as low-dimensional interpolation [74]."

Mechanistic interpretability (MI)

- Tegmark suggests that algorithms can be distilled from neural networks by a kind of computational neuroscience
- "Once a good algorithm has been discovered, neural networks lose their computational advantage. Indeed, PCCs may be more efficient, because neural networks have been shown to underperform GOFAI methods on basic tasks such as low-dimensional interpolation [74]."
  - In my opinion, this is an understatement. There are very few tasks where neural networks are even close to computationally optimal for the tasks they have been trained to do, and tailor-made programs can clearly win by orders of magnitude

Mechanistic interpretability (MI)

- Tegmark suggests that algorithms can be distilled from neural networks by a kind of computational neuroscience
- "Once a good algorithm has been discovered, neural networks lose their computational advantage. Indeed, PCCs may be more efficient, because neural networks have been shown to underperform GOFAI methods on basic tasks such as low-dimensional interpolation [74]."
  - In my opinion, this is an understatement. There are very few tasks where neural networks are even close to computationally optimal for the tasks they have been trained to do, and tailor-made programs can clearly win by orders of magnitude
  - I have serious ethical concerns about the increasingly extreme levels of compute being used on ML of late, centralizing power in the few companies that can afford it

Mechanistic interpretability (MI)

- Tegmark suggests that algorithms can be distilled from neural networks by a kind of computational neuroscience
- "Once a good algorithm has been discovered, neural networks lose their computational advantage. Indeed, PCCs may be more efficient, because neural networks have been shown to underperform GOFAI methods on basic tasks such as low-dimensional interpolation [74]."
  - In my opinion, this is an understatement. There are very few tasks where neural networks are even close to computationally optimal for the tasks they have been trained to do, and tailor-made programs can clearly win by orders of magnitude
  - I have serious ethical concerns about the increasingly extreme levels of compute being used on ML of late, centralizing power in the few companies that can afford it
- We *should* be using MI to get simple algorithms out of the deeply embedded gym environment used for the discovery process

Mechanistic interpretability (MI)

- Tegmark suggests that algorithms can be distilled from neural networks by a kind of computational neuroscience
- "Once a good algorithm has been discovered, neural networks lose their computational advantage. Indeed, PCCs may be more efficient, because neural networks have been shown to underperform GOFAI methods on basic tasks such as low-dimensional interpolation [74]."
  - In my opinion, this is an understatement. There are very few tasks where neural networks are even close to computationally optimal for the tasks they have been trained to do, and tailor-made programs can clearly win by orders of magnitude
  - I have serious ethical concerns about the increasingly extreme levels of compute being used on ML of late, centralizing power in the few companies that can afford it
- We *should* be using MI to get simple algorithms out of the deeply embedded gym environment used for the discovery process
- Possibly this ends up just looking like traditional program synthesis?

Mechanistic interpretability (MI)

- Tegmark suggests that algorithms can be distilled from neural networks by a kind of computational neuroscience
- "Once a good algorithm has been discovered, neural networks lose their computational advantage. Indeed, PCCs may be more efficient, because neural networks have been shown to underperform GOFAI methods on basic tasks such as low-dimensional interpolation [74]."
  - In my opinion, this is an understatement. There are very few tasks where neural networks are even close to computationally optimal for the tasks they have been trained to do, and tailor-made programs can clearly win by orders of magnitude
  - I have serious ethical concerns about the increasingly extreme levels of compute being used on ML of late, centralizing power in the few companies that can afford it
- We *should* be using MI to get simple algorithms out of the deeply embedded gym environment used for the discovery process
- Possibly this ends up just looking like traditional program synthesis?
- This is also just a fancy kind of hyperparameter tuning

Regulating AGI

- A key component of this plan is that regulators ensure that components are proved correct for safety reasons
- This is important because formal proofs at this level are not economically justifiable otherwise
  - Intel might try to prove their chips correct today, but certainly Linux wouldn't. Who would pay for it?

Regulating AGI

- A key component of this plan is that regulators ensure that components are proved correct for safety reasons
- This is important because formal proofs at this level are not economically justifiable otherwise
  - Intel might try to prove their chips correct today, but certainly Linux wouldn't. Who would pay for it?
- My own utopian vision is that we should make formal proof more economically viable by making it easier to use, esp. in software development

Regulating AGI

- A key component of this plan is that regulators ensure that components are proved correct for safety reasons
- This is important because formal proofs at this level are not economically justifiable otherwise
  - Intel might try to prove their chips correct today, but certainly Linux wouldn't. Who would pay for it?
- My own utopian vision is that we should make formal proof more economically viable by making it easier to use, esp. in software development
- Regulations require literally every government to be in agreement on this topic, like a nuclear arms pact. This never seems to work in practice, and it's even easier here to have rogue actors

Regulating AGI

- A key component of this plan is that regulators ensure that components are proved correct for safety reasons
- This is important because formal proofs at this level are not economically justifiable otherwise
  - Intel might try to prove their chips correct today, but certainly Linux wouldn't. Who would pay for it?
- My own utopian vision is that we should make formal proof more economically viable by making it easier to use, esp. in software development
- Regulations require literally every government to be in agreement on this topic, like a nuclear arms pact. This never seems to work in practice, and it's even easier here to have rogue actors
- I think the best we can hope for is increased use of formal proof in safety-critical systems

How to write correct programs

Verified compilation

- Goal: To have a way to write programs with functional correctness properties

Verified compilation

- Goal: To have a way to write programs with functional correctness properties
- Goal: Minimize the collection of unverified components (the TCB)

Verified compilation

- Goal: To have a way to write programs with functional correctness properties
- Goal: Minimize the collection of unverified components (the TCB)
- Goal: Minimize the amount of labor involved in the previous goals

Verified compilation

- Goal: To have a way to write programs with functional correctness properties
- Goal: Minimize the collection of unverified components (the TCB)
- Goal: Minimize the amount of labor involved in the previous goals
  - both human and machine labor

Verified compilation

- Goal: To have a way to write programs with functional correctness properties
- Goal: Minimize the collection of unverified components (the TCB)
- Goal: Minimize the amount of labor involved in the previous goals
  - both human and machine labor
  - human labor: limits scalability because it makes it too costly to build verified systems

Verified compilation

- Goal: To have a way to write programs with functional correctness properties
- Goal: Minimize the collection of unverified components (the TCB)
- Goal: Minimize the amount of labor involved in the previous goals
  - both human and machine labor
  - human labor: limits scalability because it makes it too costly to build verified systems
  - computer labor: limits scalability less making it costly to maintain verified systems, but mostly limits *availability* to users without big computing budgets

Verified compilation

- Solution: Raise the level of abstraction

Verified compilation

- Solution: Raise the level of abstraction
  - applies to both
    programs (compilers for high level languages)
    and proofs (proof assistants and verified compilers)

Verified compilation

- Solution: Raise the level of abstraction
  - applies to both
    programs (compilers for high level languages)
    and proofs (proof assistants and verified compilers)
- How can we actually build tools to facilitate this?

Verified compilation

- Solution: Raise the level of abstraction
  - applies to both
    programs (compilers for high level languages)
    and proofs (proof assistants and verified compilers)
- How can we actually build tools to facilitate this?
- Interlude: CompCert[1]

[1] https://www.cs.yale.edu/flint/cs421/lectureNotes/Fall12/compcert-slides.pdf

Verified compilation

- Solution: Raise the level of abstraction
  - applies to both
    programs (compilers for high level languages)
    and proofs (proof assistants and verified compilers)
- How can we actually build tools to facilitate this?
- Interlude: CompCert[1]
- Interlude: Metamath C

[1] https://www.cs.yale.edu/flint/cs421/lectureNotes/Fall12/compcert-slides.pdf

Metamath C

- Experimental language and compiler for writing verified programs
- Syntax somewhat similar to Dafny or Rust
- What sets it apart from most other programming languages is that *the compiler outputs a proof*

Two perspectives on provable correctness

Low level

- Computers execute machine code. My computer executes x86, so we need the execution semantics of x86

Two perspectives on provable correctness

Low level

- Computers execute machine code. My computer executes x86, so we need the execution semantics of x86
  - Read the spec (this is public information)

Two perspectives on provable correctness

Low level

- Computers execute machine code. My computer executes x86, so we need the execution semantics of x86
  - Read the spec (this is public information)
- We also need the specification of the input and output relation; for linux this means specifying how the `read()` and `write()` system calls work

Two perspectives on provable correctness

Low level

- Computers execute machine code. My computer executes x86, so we need the execution semantics of x86
  - Read the spec (this is public information)
- We also need the specification of the input and output relation; for linux this means specifying how the `read()` and `write()` system calls work
  - Read the... manpage? (this is mostly spec'd in POSIX, although not very well)

Two perspectives on provable correctness

Low level

- Computers execute machine code. My computer executes x86, so we need the execution semantics of x86
  - Read the spec (this is public information)
- We also need the specification of the input and output relation; for linux this means specifying how the `read()` and `write()` system calls work
  - Read the... manpage? (this is mostly spec'd in POSIX, although not very well)
- Now we can say things like "This sequence of bytes of machine code, when executed, will calculate the following input-output relation"

Two perspectives on provable correctness

Low level

- Computers execute machine code. My computer executes x86, so we need the execution semantics of x86
  - Read the spec (this is public information)
- We also need the specification of the input and output relation; for linux this means specifying how the `read()` and `write()` system calls work
  - Read the... manpage? (this is mostly spec'd in POSIX, although not very well)
- Now we can say things like "This sequence of bytes of machine code, when executed, will calculate the following input-output relation"
  - Good enough for one-shot programs, but programs with an interaction component need more exotic formalization

Two perspectives on provable correctness

High level

- Proof languages like Lean generally operate at the level of a *type system*, often an elaborate one

Two perspectives on provable correctness

High level

- Proof languages like Lean generally operate at the level of a *type system*, often an elaborate one
- The purpose is to model abstract entities that the programmer wants to reason about

Two perspectives on provable correctness

High level

- Proof languages like Lean generally operate at the level of a *type system*, often an elaborate one
- The purpose is to model abstract entities that the programmer wants to reason about
- Proofs are done at this level

Two perspectives on provable correctness

High level

- Proof languages like Lean generally operate at the level of a *type system*, often an elaborate one
- The purpose is to model abstract entities that the programmer wants to reason about
- Proofs are done at this level
- For systems languages like C or Rust, the type system is also fairly transparent about how types relate to bytes (data representation)

Two perspectives on provable correctness

High level

- Proof languages like Lean generally operate at the level of a *type system*, often an elaborate one
- The purpose is to model abstract entities that the programmer wants to reason about
- Proofs are done at this level
- For systems languages like C or Rust, the type system is also fairly transparent about how types relate to bytes (data representation)

The role of a compiler is to connect the high level view to the low level, ideally with minimal intervention from the user

Type system as prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable

Type system as prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs

Type system as prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs
- Java: Generic types, type polymorphism

Type system as prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs
- Java: Generic types, type polymorphism
- Haskell: Algebraic data types

Type system as prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs
- Java: Generic types, type polymorphism
- Haskell: Algebraic data types
- Rust: Linear types

Type system as prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs
- Java: Generic types, type polymorphism
- Haskell: Algebraic data types
- Rust: Linear types
- Static analyzers: Value analysis, contract checking

Type system as prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs
- Java: Generic types, type polymorphism
- Haskell: Algebraic data types
- Rust: Linear types
- Static analyzers: Value analysis, contract checking
- Lean: Dependent types, proof objects, the type system is an ITP

Type system as prover

Programming languages have come a long way in terms of provable correctness

- Assembly: bare minimum type system required to make instructions compilable
- C: Typed pointers, structs
- Java: Generic types, type polymorphism
- Haskell: Algebraic data types
- Rust: Linear types
- Static analyzers: Value analysis, contract checking
- Lean: Dependent types, proof objects, the type system is an ITP
- F*: the type system is an ATP

A type checker is just a simple theorem prover; the study of one naturally leads to the other

Linear types

- In a traditional type system, any value in the context can be used any number of times

Linear types

- In a traditional type system, any value in the context can be used any number of times
- In a linear type system, values are "used up" and so can be used to represent non-renewable resources and ownership

Linear types

- In a traditional type system, any value in the context can be used any number of times
- In a linear type system, values are "used up" and so can be used to represent non-renewable resources and ownership
- (It really helps if you have written Rust code to understand how this works)

Linear types

- In a traditional type system, any value in the context can be used any number of times
- In a linear type system, values are “used up” and so can be used to represent non-renewable resources and ownership
- (It really helps if you have written Rust code to understand how this works)

```
let vec: Vec<u32> = vec![0, 1, 2];
drop(vec);
drop(vec); // Error!
```

Separation logic

- Once you start caring about values and not just types, propositions become linear types, and computation can consume propositions
  - $\{x \mapsto 1\}\ x := 2\ \{x \mapsto 2\}$

Separation logic

- Once you start caring about values and not just types, propositions become linear types, and computation can consume propositions
  - $\{x \mapsto 1\}\ x := 2\ \{x \mapsto 2\}$
- This is necessary to avoid everything becoming parameterized on the machine state
  - These propositions are modeled internally as `State -> Prop`

Separation logic

- Once you start caring about values and not just types, propositions become linear types, and computation can consume propositions
  - $\{x \mapsto 1\}\ x := 2\ \{x \mapsto 2\}$
- This is necessary to avoid everything becoming parameterized on the machine state
  - These propositions are modeled internally as `State -> Prop`
- Separation logic adds the ability to refer to *pieces* of the state, with a structural rule saying that unrelated parts of the state are unchanged
  - $\{y \mapsto v * z \mapsto w * x \mapsto 1\}\ x := 2\ \{y \mapsto v * z \mapsto w * x \mapsto 2\}$

Separation logic

- Once you start caring about values and not just types, propositions become linear types, and computation can consume propositions
  - $\{x \mapsto 1\}\ x := 2\ \{x \mapsto 2\}$
- This is necessary to avoid everything becoming parameterized on the machine state
  - These propositions are modeled internally as `State -> Prop`
- Separation logic adds the ability to refer to *pieces* of the state, with a structural rule saying that unrelated parts of the state are unchanged
  - $\{y \mapsto v * z \mapsto w * x \mapsto 1\}\ x := 2\ \{y \mapsto v * z \mapsto w * x \mapsto 2\}$
  - This is very important for modularity

Metamath C

- "C with dependent types"
- Basic structure is similar to C
- "hypothesis variables" hold on to separating propositions
- computationally irrelevant (ghost) variables are marked

$it \in \text{Item} ::= \mathsf{type}\ S(\overline{\alpha}, \overline{R}) := \tau$	type declaration
$\mid \mathsf{const}\ t := e$	constant declaration
$\mid \mathsf{global}\ t := e$	global variable declaration
$\mid \mathsf{proc}\ f(\overline{R}) : \overline{R} := e$	procedure declaration
$e \in \text{Expr} ::= x$	variable reference
$\mid () \mid \mathsf{true} \mid \mathsf{false} \mid n$	constants
$\mid e_1 + e_2 \mid e_1 * e_2 \mid -e$	addition, multiplication, negation
$\mid \mathsf{if}\ h^? : e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$	conditionals
$\mid \langle \overline{e} \rangle$	tuple
$\mid \mathsf{let}\ t := e_1\ \mathsf{in}\ e_2$	assignment to a variable
$\mid \eta \leftarrow e;\ e$	assignment
$\mid F(\overline{e})$	procedure call
$\mid \mathsf{return}\ \overline{e}$	procedure return
$\mid \mathsf{label}\ \overline{k(\overline{R}) := e}\ \mathsf{in}\ e'$	local mutual tail recursion
$\mid \mathsf{goto}\ k(\overline{e})$	local tail call
$\mid \mathsf{unreachable}\ e$	unreachable statement
$\mid \mathsf{entail}\ \overline{e}\ p$	entailment proof
$\mid \mathsf{assert}\ e$	assertion
$\mid \mathsf{typeof}\ e$	take the type of a variable
$\mid \dots$	

Metamath C

- Types are a combination of C/Rust-style types, and separating propositions
- A variable can be "moved" by using it (substructural logic)
- The typeof operator can "take" the type of a variable $x : \tau$ and put it in a hypothesis $h : \boxed{x : \tau}$, and pun puts it back
- Not pictured: match, ghost, owned/shared/mutable pointers, heap references, while/for loops, variants and invariants, ...

$t \in \text{TupPat} ::= \_ \mid x \mid \boxed{x}$	ignored, variable, ghost variable
$\mid t : \tau \mid \langle \overline{t} \rangle$	type ascription, tuple
$R \in \text{Arg} ::= x : \tau \mid \boxed{x} : \tau$	regular/ghost argument
$\tau \in \text{Type} ::= \top \mid \bot \mid \mathbf{1} \mid \mathsf{bool}$	true, false, unit, booleans
$\mid \alpha$	type variable
$\mid S(\overline{\tau}, \overline{e})$	user-defined type
$\mid \mathbb{N}_s \mid \mathbb{Z}_s$	unsigned/signed integers
$\mid [\tau;\ e]$	array type
$\mid \divideontimes \overline{\tau} \mid \sum \overline{R}$	(dependent) tuple type
$\mid \tau^?$	maybe-uninit type
$\mid e$	assert that a boolean value is true
$\mid \forall x : \tau,\ \tau' \mid \exists x : \tau,\ \tau'$	universal, existential quantification
$\mid \tau_1 \wedge \tau_2 \mid \tau_1 \vee \tau_2$	conjunction, disjunction
$\mid \tau_1 \to \tau_2 \mid \neg\tau$	implication, negation
$\mid \tau_1 * \tau_2 \mid \tau_1 \mathrel{-\!\!*} \tau_2$	separating conjunction/implication
$\mid \mathsf{ref}^a\ \tau$	borrowed type
$\mid \&^{sn}\ e$	pointer type
$\mid e \mapsto e'$	points-to assertion
$\mid \boxed{x : \tau}$	typing assertion
$\mid \boxed{\tau}$	ghost type
$\mid \lvert\tau\rvert$	moved type

Metamath C: is_prime

Here is an example[2] of a simple function to compute primes:

```
proc is_prime (n: u32) : bool :=
  for i in 2..n-1 do
    if n % i = 0 then
      return false
  true
```

[2]MMC uses a lisp-like syntax that I have not grown to like, so this is an artist's interpretation

Metamath C: is_prime

With proofs:

```
def decidable (p: Type) := (b: bool) * (b ↔ p)
proc is_prime (n: u32) (h: n ≠ 0) :
  decidable (n = 1 ∨ prime n) :=
  for h2: i in 2..n-1 do
    if h3: n % i = 0 then
      -- h2: i ∈ 2..n-1
      -- h3: n % i = 0
      return (false,
        show false ↔ n = 1 ∨ prime n
        ...)
  (true,
    show true ↔ n = 1 ∨ prime n
    ...)
```

Metamath C: is_prime

We need more to prove the omitted parts:

```
def decidable (p: Type) := (b: bool) * (b ↔ p)
proc is_prime (n: u32) (h: n ≠ 0) :
  decidable (n = 1 ∨ prime n) :=
  for h2: i in 2..n-1 do
    show n % i ≠ 0
    if h3: n % i = 0 then
      -- h2: i ∈ 2..n-1
      -- h3: n % i = 0
      return (false, ...)
    else
      -- h3: n % i ≠ 0
      h3
  -- h2: ∀ i ∈ 2..n-1, n % i ≠ 0
  (true, ...)
```

Metamath C: Mutation

```
proc _ :=
  let x: u8 := 1
  have x = 1 := rfl
  let y := &x -- y: &sn x
  let z := *y -- z: u8 := x
  have z = 1 := rfl
  *y <- 2
  have z = 1 := rfl
  have x = 2 := rfl
  -- x† := 1
  -- y := &sn x†
  -- z := x†
  -- x := 2
```

Metamath C: Mutation

It's not just shadowing:

```
proc _ (b: bool) :=
  let x: u8 := 1
  have h: x = 1 := rfl
  if b then
    x <- 2
  have x = 1 := rfl -- fails
  -- x† := 1
  -- h: x† = 1
  -- x: u8
```

Metamath C: Mutation

Carrying mutation information out of an if statement:

```
proc _ (b: bool) :=
  let x: u8 := 1
  have h: x = 1 ∨ x = 2 := or.inl rfl
  if b then
    x <- 2
    h: x = 1 ∨ x = 2 <- or.inr rfl
  -- x: u8
  -- h: x = 1 ∨ x = 2
```

Metamath C: Big integers

```
proc _ : u64 :=
  let x: nat := 10 ^ 60 + 1 -- not representable
  x as u64 -- representable!
```

- There are types `u8`, `u16`, `u32`, `u64`, `nat`
- There is no big integer implementation, `nat` means *unbounded* integers and maps to the “true” $\mathbb{N}$
- Compiler will fail if it can’t figure out how to compile your expression
- Usable in ghost variables and specifications:

```
proc _ : (r: u64) * (ghost x: nat) * (x as u64 = r) :=
  let x: nat := 10 ^ 60 + 1
  (x as u64, x, rfl)
```

Metamath C: Pointers

```
proc _ (x: &u64) : u64 :=
  let (v, ptr) := x
  -- v: ref u64
  -- ptr: &sn v
  let y := *ptr
  -- y := v
  -- *ptr <- 2 -- fails
  y
```

Metamath C: Ghost state

```
proc _ (x: u64) (ghost y: u64) : u64 :=
  -- x: u64
  -- ghost y: u64
  let z := y
  -- ghost z := y
  z -- Error, returning ghost data in relevant position

proc _ (x: u64) (ghost y: u64) : u64 := y -- not ok
proc _ (x: u64) (ghost y: u64) : u64 := x -- ok
proc _ (x: u64) (ghost y: u64) : ghost u64 := y -- ok
```

- Types can refer to ghost values
- `ghost y: u64` is equivalent to `y: ghost u64`

Getting proofs out of a compiler

Approaches to verified compilation

- In CompCert, there is a theorem of the following form:

Theorem (CompCert correctness)

If the compiler `compile` *is run on input program P to get assembly program Q, and every possible behavior of P under the execution semantics of C is not UB* (bad)*, then the assembly program Q exhibits only behaviors of the original program P. That is:*

$$\forall P\ Q,\ \ \texttt{compile}(P) = \mathsf{OK}(Q) \rightarrow$$
$$(\forall \sigma, \texttt{exec\_C}(P, \sigma) \rightarrow \neg\mathsf{bad}(\sigma)) \rightarrow$$
$$(\forall \sigma, \texttt{exec\_asm}(Q, \sigma) \rightarrow \texttt{exec\_C}(P, \sigma))$$

Approaches to verified compilation

- The Metamath C compiler generates theorems of the following form:

Theorem (Metamath C correctness)

Every behavior exhibited by machine code program `myprog` *(does not exhibit UB and) satisfies specification* `myspec`*:*

$$\forall \sigma, \mathtt{exec\_x86}(\mathtt{myprog}, \sigma) \to \neg \mathtt{bad}(\sigma) \land \mathtt{myspec}(\sigma)$$

Approaches to verified compilation

Some key differences:

- The CompCert theorem is generalized over possible programs P and Q, while the Metamath C theorem is specialized to an individual program $Q :=$ `myprog` which is generated by the compiler
- The CompCert theorem does not make any mention of user specifications, only the C specification. The user is still responsible for proving the program meets its specification via some C analysis framework like VST

Approaches to verified compilation

Arguably, the user's goal is to produce something like the Metamath C theorem. Let us make the CompCert theorem look more like it:

1. First, we write a program `my_C_prog` that is correct
2. We prove it is correct (in Coq), yielding a theorem $\texttt{my\_C\_prog} \models \texttt{myspec}$

$$P \models S \iff \forall \sigma, \texttt{exec\_C}(P, \sigma) \to \neg \mathsf{bad}(\sigma) \land S(\sigma)$$

3. We run the CompCert compiler (in Coq), yielding a term `asm_out` and a proof of $\texttt{compile}(\texttt{my\_C\_prog}) = \mathsf{OK}(\texttt{asm\_out})$
4. By composing with the correctness theorem we obtain $\forall \sigma, \texttt{exec\_asm}(\texttt{asm\_out}, \sigma) \to \neg \mathsf{bad}(\sigma) \land \texttt{myspec}(\sigma)$

Approaches to verified compilation

Most of these steps have an analogue in the Metamath C model:

1. First, we write a program `my_MMC_prog` that is correct
2. Because `my_MMC_prog` is written in a language with proofs we can simultaneously prove it is correct
3. We run the Metamath C compiler, yielding a term `asm_out` and a proof of $\forall\sigma, \texttt{exec\_x86}(\texttt{x86\_out}, \sigma) \to \neg\texttt{bad}(\sigma) \land \texttt{myspec}(\sigma)$

Approaches to verified compilation

1. The Metamath C approach is called “proof-carrying code” (PCC) in the literature
2. CompCert is a *verified compiler*, MMC is a *certifying compiler*

Approaches to verified compilation

1. The Metamath C approach is called "proof-carrying code" (PCC) in the literature
2. CompCert is a *verified compiler*, MMC is a *certifying compiler*
3. CompCert also makes use of translation validation: run an unverified program and validate the results

Translation Validation

```
(** [regalloc] is the external register allocator.
  It is written in OCaml in [backend/Regalloc.ml]. *)
Parameter regalloc: RTL.function -> res LTL.function.

(** Register allocation followed by validation. *)
Definition transf_function (f: RTL.function) :
    res LTL.function :=
  match type_function f with
  | Error m => Error m
  | OK env =>
      match regalloc f with
      | Error m => Error m
      | OK tf => do x <- check_function f tf env; OK tf
      end
  end.
```

Why PCC?

- In the proof generated by CompCert, there are three parts to the proof:
  - Parts 1-2 (correctness in the source language) are written by the user and tailored to the program
  - Part 3 (evaluating the compiler) is run in the Coq kernel and depends on the program
  - Part 4 (applying the correctness theorem) is O(1) work, not program dependent
- The MMC approach omits steps 1-2 entirely from the proof and combines 3-4
- The key observation is that "evaluating the compiler" on a particular program could be assembling a correctness proof for no added cost
  - Whether "no added cost" is true depends on the performance characteristics of the kernel

Benefits of PCC

- The biggest one: the compiler doesn't have to be verified or written in a proof assistant
  - This is true also of the TV approach, but...

Benefits of PCC

- The biggest one: the compiler doesn't have to be verified or written in a proof assistant
  - This is true also of the TV approach, but...
  - Now how are we supposed to prove `compile`(`my_C_prog`) = OK(`asm_out`)?
  - `Parameter` is a synonym of `Axiom` and blocks reduction in the type theory
  - The proof can't be `refl` anymore without kernel magic

Benefits of PCC

- The biggest one: the compiler doesn't have to be verified or written in a proof assistant
  - This is true also of the TV approach, but...
  - Now how are we supposed to prove `compile(my_C_prog) = OK(asm_out)`?
  - `Parameter` is a synonym of `Axiom` and blocks reduction in the type theory
  - The proof can't be `refl` anymore without kernel magic
- Supports both deterministic and nondeterministic compilation strategies
  - "Nondeterministic" here means that the proof itself doesn't have to care how a decision was made, e.g. stack slots in a function
- The overhead of proving that a nondeterministic program can evaluate to a result is usually less than proving that a deterministic program computes a result

Scaling up?

Scaling up?

- Sorry, not yet

Scaling up?

- Sorry, not yet
- CompCert has scaled up enough to be a commercial product
- I'm interested in making sure that:

Scaling up?

- Sorry, not yet
- CompCert has scaled up enough to be a commercial product
- I'm interested in making sure that:
  - the proofs generated are small

Scaling up?

- Sorry, not yet
- CompCert has scaled up enough to be a commercial product
- I'm interested in making sure that:
  - the proofs generated are small
  - the proofs are fast to produce and to check

Scaling up?

- Sorry, not yet
- CompCert has scaled up enough to be a commercial product
- I'm interested in making sure that:
  - the proofs generated are small
  - the proofs are fast to produce and to check
  - they rely on a minimum of supporting material

Scaling up?

- Sorry, not yet
- CompCert has scaled up enough to be a commercial product
- I'm interested in making sure that:
  - the proofs generated are small
  - the proofs are fast to produce and to check
  - they rely on a minimum of supporting material
  - "I paid the price so you don't have to": provide generous affordances for inefficient proofs e.g. ATP

Scaling up?

- Sorry, not yet
- CompCert has scaled up enough to be a commercial product
- I'm interested in making sure that:
  - the proofs generated are small
  - the proofs are fast to produce and to check
  - they rely on a minimum of supporting material
  - "I paid the price so you don't have to": provide generous affordances for inefficient proofs e.g. ATP
  - use an "embeddable logic" so that it is compatible with many targets

Scaling up?

- Sorry, not yet
- CompCert has scaled up enough to be a commercial product
- I'm interested in making sure that:
  - the proofs generated are small
  - the proofs are fast to produce and to check
  - they rely on a minimum of supporting material
  - "I paid the price so you don't have to": provide generous affordances for inefficient proofs e.g. ATP
  - use an "embeddable logic" so that it is compatible with many targets
- I am often concerned by academic projects that do a thing but clearly aren't designed to scale 100x, because that's where the market actually is

Scaling up?

- Sorry, not yet
- CompCert has scaled up enough to be a commercial product
- I'm interested in making sure that:
  - the proofs generated are small
  - the proofs are fast to produce and to check
  - they rely on a minimum of supporting material
  - "I paid the price so you don't have to": provide generous affordances for inefficient proofs e.g. ATP
  - use an "embeddable logic" so that it is compatible with many targets
- I am often concerned by academic projects that do a thing but clearly aren't designed to scale 100x, because that's where the market actually is
  - ...Then again, I'm a small fish in a big pond: MMC is basically vaporware by comparison to Dafny, Why3, Verus, Lean, Bedrock

Conclusion

- Tegmark and Omohundro have some interesting ideas, many of which line up with my grand visions of the world

Conclusion

- Tegmark and Omohundro have some interesting ideas, many of which line up with my grand visions of the world
- but some of the goals and methods to achieve the goals look too naive

Conclusion

- Tegmark and Omohundro have some interesting ideas, many of which line up with my grand visions of the world
- but some of the goals and methods to achieve the goals look too naive
- In a way it doesn't matter, the intermediate goals are hugely useful anyway

Conclusion

- Tegmark and Omohundro have some interesting ideas, many of which line up with my grand visions of the world
- but some of the goals and methods to achieve the goals look too naive
- In a way it doesn't matter, the intermediate goals are hugely useful anyway
- The performance issues of ML are standing in the way of the singularity in a very literal sense

Conclusion

- Tegmark and Omohundro have some interesting ideas, many of which line up with my grand visions of the world
- but some of the goals and methods to achieve the goals look too naive
- In a way it doesn't matter, the intermediate goals are hugely useful anyway
- The performance issues of ML are standing in the way of the singularity in a very literal sense
- Pivoting from pure ML to MI-based programs makes it much more practical to use ML training in resource constrained environments like ITP users' laptops or ATP splitting heuristics

Conclusion

- Tegmark and Omohundro have some interesting ideas, many of which line up with my grand visions of the world
- but some of the goals and methods to achieve the goals look too naive
- In a way it doesn't matter, the intermediate goals are hugely useful anyway
- The performance issues of ML are standing in the way of the singularity in a very literal sense
- Pivoting from pure ML to MI-based programs makes it much more practical to use ML training in resource constrained environments like ITP users' laptops or ATP splitting heuristics
- Verified compilation is a practical method for producing programs with strong correctness properties. We need to work to make it more mainstream. I don't think we need "AI" here, just better languages

Conclusion

- Tegmark and Omohundro have some interesting ideas, many of which line up with my grand visions of the world
- but some of the goals and methods to achieve the goals look too naive
- In a way it doesn't matter, the intermediate goals are hugely useful anyway
- The performance issues of ML are standing in the way of the singularity in a very literal sense
- Pivoting from pure ML to MI-based programs makes it much more practical to use ML training in resource constrained environments like ITP users' laptops or ATP splitting heuristics
- Verified compilation is a practical method for producing programs with strong correctness properties. We need to work to make it more mainstream. I don't think we need "AI" here, just better languages
  - But I won't say no to a program that does my proof for me!

Conclusion

- Tegmark and Omohundro have some interesting ideas, many of which line up with my grand visions of the world
- but some of the goals and methods to achieve the goals look too naive
- In a way it doesn't matter, the intermediate goals are hugely useful anyway
- The performance issues of ML are standing in the way of the singularity in a very literal sense
- Pivoting from pure ML to MI-based programs makes it much more practical to use ML training in resource constrained environments like ITP users' laptops or ATP splitting heuristics
- Verified compilation is a practical method for producing programs with strong correctness properties. We need to work to make it more mainstream. I don't think we need "AI" here, just better languages
  - But I won't say no to a program that does my proof for me!
  - ... as long as it takes less time than me and doesn't require me to sell my soul to OpenAI