Generalised Computability and Complexity in Set-Theoretic Contexts

Desmond Lau

13 March 2024

Outline

1. New Notions of Generalised (Relative) Computability
   - Computation on Arbitrary Sets
   - Restricting Abstract State Machines
   - The Result
2. Degrees of Small Extensions and Complexity of Local Methods
   - Degrees of Small Extensions
   - Local Method Definitions
   - Results
3. Conclusion

Outline

1. New Notions of Generalised (Relative) Computability
   - Computation on Arbitrary Sets
   - Restricting Abstract State Machines
   - The Result
2. Degrees of Small Extensions and Complexity of Local Methods
   - Degrees of Small Extensions
   - Local Method Definitions
   - Results
3. Conclusion

Conventional Computations

- Turing machines are the de facto abstract models.
- Inputs and outputs of computations are typically finite strings.
- These strings can be coded as natural numbers, and so have hereditarily finite set-theoretic representations.
- A computation must terminate in finite time for them to have an output.

Conventional Computations

- Turing machines are the de facto abstract models.
- Inputs and outputs of computations are typically finite strings.
- These strings can be coded as natural numbers, and so have hereditarily finite set-theoretic representations.
- A computation must terminate in finite time for them to have an output.

Conventional Computations

- Turing machines are the de facto abstract models.
- Inputs and outputs of computations are typically finite strings.
- These strings can be coded as natural numbers, and so have hereditarily finite set-theoretic representations.
- A computation must terminate in finite time for them to have an output.

Conventional Computations

- Turing machines are the de facto abstract models.
- Inputs and outputs of computations are typically finite strings.
- These strings can be coded as natural numbers, and so have hereditarily finite set-theoretic representations.
- A computation must terminate in finite time for them to have an output.

Conventional Computations

- Turing machines are the de facto abstract models.
- Inputs and outputs of computations are typically finite strings.
- These strings can be coded as natural numbers, and so have hereditarily finite set-theoretic representations.
- A computation must terminate in finite time for them to have an output.

Computing with Larger Objects

- What if we want to "compute" things that require more than finite information and/or time-steps?
- For example, one may ask the question

  "Is the Stone-Cech compactification of $\mathbb{R}$ computable from $\mathbb{R}$?"
- This question does not make sense if we interpret "computable" according to convention.
- We are thus motivated to extend the notion of computation to sets far larger than finite strings and natural numbers.

Computing with Larger Objects

- What if we want to "compute" things that require more than finite information and/or time-steps?
- For example, one may ask the question

  "Is the Stone-Cech compactification of $\mathbb{R}$ computable from $\mathbb{R}$?"
- This question does not make sense if we interpret "computable" according to convention.
- We are thus motivated to extend the notion of computation to sets far larger than finite strings and natural numbers.

Computing with Larger Objects

- What if we want to "compute" things that require more than finite information and/or time-steps?
- For example, one may ask the question
  "Is the Stone-Cech compactification of $\mathbb{R}$ computable from $\mathbb{R}$?"
- This question does not make sense if we interpret "computable" according to convention.
- We are thus motivated to extend the notion of computation to sets far larger than finite strings and natural numbers.

Computing with Larger Objects

- What if we want to "compute" things that require more than finite information and/or time-steps?
- For example, one may ask the question

  "Is the Stone-Cech compactification of $\mathbb{R}$ computable from $\mathbb{R}$?"
- This question does not make sense if we interpret "computable" according to convention.
- We are thus motivated to extend the notion of computation to sets far larger than finite strings and natural numbers.

Computing with Larger Objects

- What if we want to "compute" things that require more than finite information and/or time-steps?
- For example, one may ask the question

  "Is the Stone-Cech compactification of $\mathbb{R}$ computable from $\mathbb{R}$?"
- This question does not make sense if we interpret "computable" according to convention.
- We are thus motivated to extend the notion of computation to sets far larger than finite strings and natural numbers.

Abstract State Machines: History

- Gurevich introduced abstract state machines (ASMs) in 2000 as a formalisation of algorithms.
- ASMs are extremely general and flexible, and have no *a priori* size limitations.
- Captures the high-level design of an algorithm, unlike other abstract models of generalised computation.
- Found ample use in engineering, but pretty much untested in set computation.

Abstract State Machines: History

- Gurevich introduced abstract state machines (ASMs) in 2000 as a formalisation of algorithms.
- ASMs are extremely general and flexible, and have no *a priori* size limitations.
- Captures the high-level design of an algorithm, unlike other abstract models of generalised computation.
- Found ample use in engineering, but pretty much untested in set computation.

Abstract State Machines: History

- Gurevich introduced abstract state machines (ASMs) in 2000 as a formalisation of algorithms.
- ASMs are extremely general and flexible, and have no *a priori* size limitations.
- Captures the high-level design of an algorithm, unlike other abstract models of generalised computation.
- Found ample use in engineering, but pretty much untested in set computation.

Abstract State Machines: History

- Gurevich introduced abstract state machines (ASMs) in 2000 as a formalisation of algorithms.
- ASMs are extremely general and flexible, and have no *a priori* size limitations.
- Captures the high-level design of an algorithm, unlike other abstract models of generalised computation.
- Found ample use in engineering, but pretty much untested in set computation.

Abstract State Machines: History

- Gurevich introduced abstract state machines (ASMs) in 2000 as a formalisation of algorithms.
- ASMs are extremely general and flexible, and have no *a priori* size limitations.
- Captures the high-level design of an algorithm, unlike other abstract models of generalised computation.
- Found ample use in engineering, but pretty much untested in set computation.

Abstract State Machines: Definition

- An ASM comprises
  - a collection of states, including initial and final states,
  - a transition function mapping one state to another.
- Every state in an ASM is a first-order structure with a finite signature.
- The transition function of an ASM
  - cannot alter the base set of a state,
  - preserves isomorphism of states,
  - "determines" the next state based on finite information about the current state (*bounded exploration*).

Abstract State Machines: Definition

- An ASM comprises
  - a collection of states, including initial and final states,
  - a transition function mapping one state to another.
- Every state in an ASM is a first-order structure with a finite signature.
- The transition function of an ASM
  - cannot alter the base set of a state,
  - preserves isomorphism of states,
  - "determines" the next state based on finite information about the current state (*bounded exploration*).

Abstract State Machines: Definition

- An ASM comprises
  - a collection of states, including initial and final states,
  - a transition function mapping one state to another.
- Every state in an ASM is a first-order structure with a finite signature.
- The transition function of an ASM
  - cannot alter the base set of a state,
  - preserves isomorphism of states,
  - "determines" the next state based on finite information about the current state (*bounded exploration*).

Abstract State Machines: Definition

- An ASM comprises
  - a collection of states, including initial and final states,
  - a transition function mapping one state to another.
- Every state in an ASM is a first-order structure with a finite signature.
- The transition function of an ASM
  - cannot alter the base set of a state,
  - preserves isomorphism of states,
  - "determines" the next state based on finite information about the current state (*bounded exploration*).

Abstract State Machines: Definition

- An ASM comprises
  - a collection of states, including initial and final states,
  - a transition function mapping one state to another.
- Every state in an ASM is a first-order structure with a finite signature.
- The transition function of an ASM
  - cannot alter the base set of a state,
  - preserves isomorphism of states,
  - "determines" the next state based on finite information about the current state (*bounded exploration*).

Abstract State Machines: Definition

- An ASM comprises
  - a collection of states, including initial and final states,
  - a transition function mapping one state to another.
- Every state in an ASM is a first-order structure with a finite signature.
- The transition function of an ASM
  - cannot alter the base set of a state,
  - preserves isomorphism of states,
  - "determines" the next state based on finite information about the current state (*bounded exploration*).

Abstract State Machines: Definition

- An ASM comprises
  - a collection of states, including initial and final states,
  - a transition function mapping one state to another.
- Every state in an ASM is a first-order structure with a finite signature.
- The transition function of an ASM
  - cannot alter the base set of a state,
  - preserves isomorphism of states,
  - "determines" the next state based on finite information about the current state (*bounded exploration*).

Abstract State Machines: Definition

- An ASM comprises
  - a collection of states, including initial and final states,
  - a transition function mapping one state to another.
- Every state in an ASM is a first-order structure with a finite signature.
- The transition function of an ASM
  - cannot alter the base set of a state,
  - preserves isomorphism of states,
  - "determines" the next state based on finite information about the current state (*bounded exploration*).

Abstract State Machines: Definition

- An ASM comprises
  - a collection of states, including initial and final states,
  - a transition function mapping one state to another.
- Every state in an ASM is a first-order structure with a finite signature.
- The transition function of an ASM
  - cannot alter the base set of a state,
  - preserves isomorphism of states,
  - "determines" the next state based on finite information about the current state (*bounded exploration*).

Abstract State Machines: Intuition

- States are analogous to Turing machine configurations.
- An initial state corresponds to a starting configuration upon receiving an input.
- A final state corresponds to an end configuration from which the output can be read.
- A transition function parallels that of a Turing machine: it codes how the machine behaves.

Abstract State Machines: Intuition

- States are analogous to Turing machine configurations.
- An initial state corresponds to a starting configuration upon receiving an input.
- A final state corresponds to an end configuration from which the output can be read.
- A transition function parallels that of a Turing machine: it codes how the machine behaves.

Abstract State Machines: Intuition

- States are analogous to Turing machine configurations.
- An initial state corresponds to a starting configuration upon receiving an input.
- A final state corresponds to an end configuration from which the output can be read.
- A transition function parallels that of a Turing machine: it codes how the machine behaves.

Abstract State Machines: Intuition

- States are analogous to Turing machine configurations.
- An initial state corresponds to a starting configuration upon receiving an input.
- A final state corresponds to an end configuration from which the output can be read.
- A transition function parallels that of a Turing machine: it codes how the machine behaves.

Abstract State Machines: Intuition

- States are analogous to Turing machine configurations.
- An initial state corresponds to a starting configuration upon receiving an input.
- A final state corresponds to an end configuration from which the output can be read.
- A transition function parallels that of a Turing machine: it codes how the machine behaves.

An Example

- Let A and B be any two sets, and choose an infinite cardinal κ large enough such that $A \cup B \subset V_\kappa$. Define

$$s_1 := (V_\kappa; \in, A, 0)$$
$$s_2 := (V_\kappa; \in, B, 1)$$
$$\text{Set of states} := \{s_1, s_2\}$$
$$\text{Set of initial states} := \{s_1\}$$
$$\text{Transition function} := \{(s_1, s_2), (s_2, s_2)\},$$

  where A, B interpret a unary relation symbol and $0, 1$ interpret a constant symbol.
- This is a valid ASM.
- Morally, it says we can compute B from A.

An Example

- Let A and B be any two sets, and choose an infinite cardinal κ large enough such that $A \cup B \subset V_\kappa$. Define

$$s_1 := (V_\kappa; \in, A, 0)$$
$$s_2 := (V_\kappa; \in, B, 1)$$
$$\text{Set of states} := \{s_1, s_2\}$$
$$\text{Set of initial states} := \{s_1\}$$
$$\text{Transition function} := \{(s_1, s_2), (s_2, s_2)\},$$

where A, B interpret a unary relation symbol and $0, 1$ interpret a constant symbol.

- This is a valid ASM.
- Morally, it says we can compute B from A.

An Example

- Let A and B be any two sets, and choose an infinite cardinal κ large enough such that $A \cup B \subset V_\kappa$. Define

$$s_1 := (V_\kappa; \in, A, 0)$$
$$s_2 := (V_\kappa; \in, B, 1)$$
$$\text{Set of states} := \{s_1, s_2\}$$
$$\text{Set of initial states} := \{s_1\}$$
$$\text{Transition function} := \{(s_1, s_2), (s_2, s_2)\},$$

where A, B interpret a unary relation symbol and $0, 1$ interpret a constant symbol.

- This is a valid ASM.
- Morally, it says we can compute B from A.

An Example

- Let A and B be any two sets, and choose an infinite cardinal κ large enough such that $A \cup B \subset V_\kappa$. Define

$$s_1 := (V_\kappa; \in, A, 0)$$
$$s_2 := (V_\kappa; \in, B, 1)$$
$$\text{Set of states} := \{s_1, s_2\}$$
$$\text{Set of initial states} := \{s_1\}$$
$$\text{Transition function} := \{(s_1, s_2), (s_2, s_2)\},$$

  where A, B interpret a unary relation symbol and $0, 1$ interpret a constant symbol.

- This is a valid ASM.
- Morally, it says we can compute B from A.

An Example

- Let A and B be any two sets, and choose an infinite cardinal κ large enough such that $A \cup B \subset V_\kappa$. Define

$$s_1 := (V_\kappa; \in, A, 0)$$
$$s_2 := (V_\kappa; \in, B, 1)$$
$$\text{Set of states} := \{s_1, s_2\}$$
$$\text{Set of initial states} := \{s_1\}$$
$$\text{Transition function} := \{(s_1, s_2), (s_2, s_2)\},$$

  where A, B interpret a unary relation symbol and $0, 1$ interpret a constant symbol.
- This is a valid ASM.
- Morally, it says we can compute B from A.

An Example

- Let A and B be any two sets, and choose an infinite cardinal κ large enough such that $A \cup B \subset V_\kappa$. Define

$$s_1 := (V_\kappa; \in, A, 0)$$
$$s_2 := (V_\kappa; \in, B, 1)$$
$$\text{Set of states} := \{s_1, s_2\}$$
$$\text{Set of initial states} := \{s_1\}$$
$$\text{Transition function} := \{(s_1, s_2), (s_2, s_2)\},$$

  where A, B interpret a unary relation symbol and $0, 1$ interpret a constant symbol.
- This is a valid ASM.
- Morally, it says we can compute B from A.

The Main Issue

- We can then compute any set from any other set.
- In particular, we can compute any set from $\emptyset$.
- "Everything is computable."
- This is worrying because in mathematics (esp. set theory) there are things that ought to be non-constructive and non-constructible.
- Intuitively, computability is weaker than constructibility, so ASMs are too encompassing a notion for set computation.

The Main Issue

- We can then compute any set from any other set.
- In particular, we can compute any set from $\emptyset$.
- "Everything is computable."
- This is worrying because in mathematics (esp. set theory) there are things that ought to be non-constructive and non-constructible.
- Intuitively, computability is weaker than constructibility, so ASMs are too encompassing a notion for set computation.

The Main Issue

- We can then compute any set from any other set.
- In particular, we can compute any set from $\emptyset$.
- "Everything is computable."
- This is worrying because in mathematics (esp. set theory) there are things that ought to be non-constructive and non-constructible.
- Intuitively, computability is weaker than constructibility, so ASMs are too encompassing a notion for set computation.

The Main Issue

- We can then compute any set from any other set.
- In particular, we can compute any set from $\emptyset$.
- "Everything is computable."
- This is worrying because in mathematics (esp. set theory) there are things that ought to be non-constructive and non-constructible.
- Intuitively, computability is weaker than constructibility, so ASMs are too encompassing a notion for set computation.

The Main Issue

- We can then compute any set from any other set.
- In particular, we can compute any set from $\emptyset$.
- "Everything is computable."
- This is worrying because in mathematics (esp. set theory) there are things that ought to be non-constructive and non-constructible.
- Intuitively, computability is weaker than constructibility, so ASMs are too encompassing a notion for set computation.

The Main Issue

- We can then compute any set from any other set.
- In particular, we can compute any set from $\emptyset$.
- "Everything is computable."
- This is worrying because in mathematics (esp. set theory) there are things that ought to be non-constructive and non-constructible.
- Intuitively, computability is weaker than constructibility, so ASMs are too encompassing a notion for set computation.

Outline

1. New Notions of Generalised (Relative) Computability
   - Computation on Arbitrary Sets
   - Restricting Abstract State Machines
   - The Result
2. Degrees of Small Extensions and Complexity of Local Methods
   - Degrees of Small Extensions
   - Local Method Definitions
   - Results
3. Conclusion

Constraints to the Transition Function

- We want to talk about local features between two states.
- Since they are first-order structures with the same signature and base set, local features are synonymous with properties given through the truth predicate $\models$.
- We modify the predicate into $\models_2$ so that we can talk about any two such structures simultaneously.
- Every ASM has its transitions governed by a transition formula: transition from s_1 to s_2 iff $(s_1, s_2) \models_2 \phi$ for transition formula ϕ.

Constraints to the Transition Function

- We want to talk about local features between two states.
- Since they are first-order structures with the same signature and base set, local features are synonymous with properties given through the truth predicate $\models$.
- We modify the predicate into $\models_2$ so that we can talk about any two such structures simultaneously.
- Every ASM has its transitions governed by a transition formula: transition from s_1 to s_2 iff $(s_1, s_2) \models_2 \phi$ for transition formula ϕ.

Constraints to the Transition Function

- We want to talk about local features between two states.
- Since they are first-order structures with the same signature and base set, local features are synonymous with properties given through the truth predicate $\models$.
- We modify the predicate into $\models_2$ so that we can talk about any two such structures simultaneously.
- Every ASM has its transitions governed by a transition formula: transition from s_1 to s_2 iff $(s_1, s_2) \models_2 \phi$ for transition formula ϕ.

Constraints to the Transition Function

- We want to talk about local features between two states.
- Since they are first-order structures with the same signature and base set, local features are synonymous with properties given through the truth predicate $\models$.
- We modify the predicate into $\models_2$ so that we can talk about any two such structures simultaneously.
- Every ASM has its transitions governed by a transition formula: transition from s_1 to s_2 iff $(s_1, s_2) \models_2 \phi$ for transition formula ϕ.

Constraints to the Transition Function

- We want to talk about local features between two states.
- Since they are first-order structures with the same signature and base set, local features are synonymous with properties given through the truth predicate $\models$.
- We modify the predicate into $\models_2$ so that we can talk about any two such structures simultaneously.
- Every ASM has its transitions governed by a transition formula: transition from s_1 to s_2 iff $(s_1, s_2) \models_2 \phi$ for transition formula ϕ.

Constraints to the Transition Function

- We want to talk about local features between two states.
- Since they are first-order structures with the same signature and base set, local features are synonymous with properties given through the truth predicate $\models$.
- We modify the predicate into $\models_2$ so that we can talk about any two such structures simultaneously.
- Every ASM has its transitions governed by a transition formula: transition from s_1 to s_2 iff $(s_1, s_2) \models_2 \phi$ for transition formula ϕ.

Locally Defining Bounded Exploration

- Previously, bounded exploration is defined by quantifying over all states of an ASM.
- It suffices but is often not easy to verify.
- By restricting transition formulas, we can formalise bounded exploration locally.

Locally Defining Bounded Exploration

- Previously, bounded exploration is defined by quantifying over all states of an ASM.
- It suffices but is often not easy to verify.
- By restricting transition formulas, we can formalise bounded exploration locally.

Locally Defining Bounded Exploration

- Previously, bounded exploration is defined by quantifying over all states of an ASM.
- It suffices but is often not easy to verify.
- By restricting transition formulas, we can formalise bounded exploration locally.

Locally Defining Bounded Exploration

- Previously, bounded exploration is defined by quantifying over all states of an ASM.
- It suffices but is often not easy to verify.
- By restricting transition formulas, we can formalise bounded exploration locally.

Collection of States is not Arbitrary

- To shift the burden of programming to the definition of the transition formula, we streamline the definitions of other components of an ASM.
- We use the language of *theories with contraints in interpretation* (*TCIs*) to define the collection of an ASM's states.
- Basically, TCIs incorporate set bounds that are not first-order definable into first-order theories.
- They generalise a number of constructions in logic.
- Here, they are used to declare the base set, variables and parameters, all of which we require to be uniform across states.

Collection of States is not Arbitrary

- To shift the burden of programming to the definition of the transition formula, we streamline the definitions of other components of an ASM.
- We use the language of *theories with contraints in interpretation* (*TCIs*) to define the collection of an ASM's states.
- Basically, TCIs incorporate set bounds that are not first-order definable into first-order theories.
- They generalise a number of constructions in logic.
- Here, they are used to declare the base set, variables and parameters, all of which we require to be uniform across states.

Collection of States is not Arbitrary

- To shift the burden of programming to the definition of the transition formula, we streamline the definitions of other components of an ASM.
- We use the language of *theories with contraints in interpretation* (*TCIs*) to define the collection of an ASM's states.
- Basically, TCIs incorporate set bounds that are not first-order definable into first-order theories.
- They generalise a number of constructions in logic.
- Here, they are used to declare the base set, variables and parameters, all of which we require to be uniform across states.

Collection of States is not Arbitrary

- To shift the burden of programming to the definition of the transition formula, we streamline the definitions of other components of an ASM.
- We use the language of *theories with contraints in interpretation* (*TCIs*) to define the collection of an ASM's states.
- Basically, TCIs incorporate set bounds that are not first-order definable into first-order theories.
- They generalise a number of constructions in logic.
- Here, they are used to declare the base set, variables and parameters, all of which we require to be uniform across states.

Collection of States is not Arbitrary

- To shift the burden of programming to the definition of the transition formula, we streamline the definitions of other components of an ASM.
- We use the language of *theories with contraints in interpretation* (*TCIs*) to define the collection of an ASM's states.
- Basically, TCIs incorporate set bounds that are not first-order definable into first-order theories.
- They generalise a number of constructions in logic.
- Here, they are used to declare the base set, variables and parameters, all of which we require to be uniform across states.

Collection of States is not Arbitrary

- To shift the burden of programming to the definition of the transition formula, we streamline the definitions of other components of an ASM.
- We use the language of *theories with contraints in interpretation* (*TCIs*) to define the collection of an ASM's states.
- Basically, TCIs incorporate set bounds that are not first-order definable into first-order theories.
- They generalise a number of constructions in logic.
- Here, they are used to declare the base set, variables and parameters, all of which we require to be uniform across states.

Ordinal Base Sets

- We specify the base set of any state to be a limit ordinal.
- This simplification is inspired by the fact that every set can be coded as a set of ordinals.
- The base set being an ordinal also fits the basic intuition of sequential memory when one thinks about computation.

Ordinal Base Sets

- We specify the base set of any state to be a limit ordinal.
- This simplification is inspired by the fact that every set can be coded as a set of ordinals.
- The base set being an ordinal also fits the basic intuition of sequential memory when one thinks about computation.

Ordinal Base Sets

- We specify the base set of any state to be a limit ordinal.
- This simplification is inspired by the fact that every set can be coded as a set of ordinals.
- The base set being an ordinal also fits the basic intuition of sequential memory when one thinks about computation.

Ordinal Base Sets

- We specify the base set of any state to be a limit ordinal.
- This simplification is inspired by the fact that every set can be coded as a set of ordinals.
- The base set being an ordinal also fits the basic intuition of sequential memory when one thinks about computation.

Input/Output Paradigm

- The interpretation of "computing B from A" is implicit in an ASM.
- We strive to make it explicit.
- Every state interprets unary relation symbols In and Out, representing the input and output tapes respectively.
- A machine computes B from A iff it has a run with A interpreting In in an initial state and B interpreting Out in a final state.

Input/Output Paradigm

- The interpretation of "computing B from A" is implicit in an ASM.
- We strive to make it explicit.
- Every state interprets unary relation symbols In and Out, representing the input and output tapes respectively.
- A machine computes B from A iff it has a run with A interpreting In in an initial state and B interpreting Out in a final state.

Input/Output Paradigm

- The interpretation of "computing B from A" is implicit in an ASM.
- We strive to make it explicit.
- Every state interprets unary relation symbols In and Out, representing the input and output tapes respectively.
- A machine computes B from A iff it has a run with A interpreting In in an initial state and B interpreting Out in a final state.

Input/Output Paradigm

- The interpretation of "computing B from A" is implicit in an ASM.
- We strive to make it explicit.
- Every state interprets unary relation symbols In and Out, representing the input and output tapes respectively.
- A machine computes B from A iff it has a run with A interpreting In in an initial state and B interpreting Out in a final state.

Input/Output Paradigm

- The interpretation of "computing B from A" is implicit in an ASM.
- We strive to make it explicit.
- Every state interprets unary relation symbols In and Out, representing the input and output tapes respectively.
- A machine computes B from A iff it has a run with A interpreting In in an initial state and B interpreting Out in a final state.

Transfinite Runs

- Runs are usually presumed to be finite in an ASM.
- Even with our restrictions, one can easily compress a finite run into a single step.
- If we allow transfinite inputs and outputs, it makes sense to allow transfinite runs.
- The transition formula dictates what happens at successor time-steps; what about limit time-steps?

Transfinite Runs

- Runs are usually presumed to be finite in an ASM.
- Even with our restrictions, one can easily compress a finite run into a single step.
- If we allow transfinite inputs and outputs, it makes sense to allow transfinite runs.
- The transition formula dictates what happens at successor time-steps; what about limit time-steps?

Transfinite Runs

- Runs are usually presumed to be finite in an ASM.
- Even with our restrictions, one can easily compress a finite run into a single step.
- If we allow transfinite inputs and outputs, it makes sense to allow transfinite runs.
- The transition formula dictates what happens at successor time-steps; what about limit time-steps?

Transfinite Runs

- Runs are usually presumed to be finite in an ASM.
- Even with our restrictions, one can easily compress a finite run into a single step.
- If we allow transfinite inputs and outputs, it makes sense to allow transfinite runs.
- The transition formula dictates what happens at successor time-steps; what about limit time-steps?

Transfinite Runs

- Runs are usually presumed to be finite in an ASM.
- Even with our restrictions, one can easily compress a finite run into a single step.
- If we allow transfinite inputs and outputs, it makes sense to allow transfinite runs.
- The transition formula dictates what happens at successor time-steps; what about limit time-steps?

Taking Limits

- We want the definition of a limit state to depend locally on the states that come before it.
- This can again be formalised using the first-order truth predicate.
- Under this constraint, we show that taking limits wherever possible (and e.g. resetting to 0 otherwise) point-wise can simulate all other definitions of a limit state.
- We adopt this all-powerful means of defining a limit state as standard.

Taking Limits

- We want the definition of a limit state to depend locally on the states that come before it.
- This can again be formalised using the first-order truth predicate.
- Under this constraint, we show that taking limits wherever possible (and e.g. resetting to 0 otherwise) point-wise can simulate all other definitions of a limit state.
- We adopt this all-powerful means of defining a limit state as standard.

Taking Limits

- We want the definition of a limit state to depend locally on the states that come before it.
- This can again be formalised using the first-order truth predicate.
- Under this constraint, we show that taking limits wherever possible (and e.g. resetting to 0 otherwise) point-wise can simulate all other definitions of a limit state.
- We adopt this all-powerful means of defining a limit state as standard.

Taking Limits

- We want the definition of a limit state to depend locally on the states that come before it.
- This can again be formalised using the first-order truth predicate.
- Under this constraint, we show that taking limits wherever possible (and e.g. resetting to 0 otherwise) point-wise can simulate all other definitions of a limit state.
- We adopt this all-powerful means of defining a limit state as standard.

Taking Limits

- We want the definition of a limit state to depend locally on the states that come before it.
- This can again be formalised using the first-order truth predicate.
- Under this constraint, we show that taking limits wherever possible (and e.g. resetting to 0 otherwise) point-wise can simulate all other definitions of a limit state.
- We adopt this all-powerful means of defining a limit state as standard.

Outline

1. New Notions of Generalised (Relative) Computability
   - Computation on Arbitrary Sets
   - Restricting Abstract State Machines
   - The Result
2. Degrees of Small Extensions and Complexity of Local Methods
   - Degrees of Small Extensions
   - Local Method Definitions
   - Results
3. Conclusion

Restricted Abstract State Machines with Parameters

- The modifications described hitherto give rise to a type of machine we call *restricted abstract state machines with parameters* (*RASMPs*).
- The parameters here are important for technical reasons.
- One can think of them as an oracle giving access to finitely many bits that may otherwise be undefinable.

Restricted Abstract State Machines with Parameters

- The modifications described hitherto give rise to a type of machine we call *restricted abstract state machines with parameters* (*RASMPs*).
- The parameters here are important for technical reasons.
- One can think of them as an oracle giving access to finitely many bits that may otherwise be undefinable.

Restricted Abstract State Machines with Parameters

- The modifications described hitherto give rise to a type of machine we call *restricted abstract state machines with parameters* (*RASMPs*).
- The parameters here are important for technical reasons.
- One can think of them as an oracle giving access to finitely many bits that may otherwise be undefinable.

Restricted Abstract State Machines with Parameters

- The modifications described hitherto give rise to a type of machine we call *restricted abstract state machines with parameters* (*RASMPs*).
- The parameters here are important for technical reasons.
- One can think of them as an oracle giving access to finitely many bits that may otherwise be undefinable.

Relative Computability Relations

- $B \leq_{\kappa}^{P,s} A$ iff some RASMP with base set κ computes B from A in less than κ time-steps.
- $B \leq_{\kappa}^{P} A$ iff some RASMP with base set κ computes B from A.
- $B \leq^{P} A$ iff $B \leq_{\kappa}^{P} A$ for some limit ordinal κ.
- If R is one of the above relations, then
  - R is transitive,
  - R sidesteps the need for an oracle-analogue, and
  - a set x being *R-computable* means $x \; R \; \emptyset$.

Relative Computability Relations

- $B \leq_{\kappa}^{P,s} A$ iff some RASMP with base set κ computes B from A in less than κ time-steps.
- $B \leq_{\kappa}^{P} A$ iff some RASMP with base set κ computes B from A.
- $B \leq^{P} A$ iff $B \leq_{\kappa}^{P} A$ for some limit ordinal κ.
- If R is one of the above relations, then
  - R is transitive,
  - R sidesteps the need for an oracle-analogue, and
  - a set x being *R-computable* means $x \; R \; \emptyset$.

Relative Computability Relations

- $B \leq_{\kappa}^{P,s} A$ iff some RASMP with base set κ computes B from A in less than κ time-steps.
- $B \leq_{\kappa}^{P} A$ iff some RASMP with base set κ computes B from A.
- $B \leq^{P} A$ iff $B \leq_{\kappa}^{P} A$ for some limit ordinal κ.
- If R is one of the above relations, then
  - R is transitive,
  - R sidesteps the need for an oracle-analogue, and
  - a set x being *R-computable* means $x \; R \; \emptyset$.

Relative Computability Relations

- $B \leq_{\kappa}^{P,s} A$ iff some RASMP with base set κ computes B from A in less than κ time-steps.
- $B \leq_{\kappa}^{P} A$ iff some RASMP with base set κ computes B from A.
- $B \leq^{P} A$ iff $B \leq_{\kappa}^{P} A$ for some limit ordinal κ.
- If R is one of the above relations, then
  - R is transitive,
  - R sidesteps the need for an oracle-analogue, and
  - a set x being *R-computable* means $x \; R \; \emptyset$.

Relative Computability Relations

- $B \leq_{\kappa}^{P,s} A$ iff some RASMP with base set κ computes B from A in less than κ time-steps.
- $B \leq_{\kappa}^{P} A$ iff some RASMP with base set κ computes B from A.
- $B \leq^{P} A$ iff $B \leq_{\kappa}^{P} A$ for some limit ordinal κ.
- If R is one of the above relations, then
  - R is transitive,
  - R sidesteps the need for an oracle-analogue, and
  - a set x being *R-computable* means $x \; R \; \emptyset$.

Relative Computability Relations

- $B \leq^{P,s}_{\kappa} A$ iff some RASMP with base set κ computes B from A in less than κ time-steps.
- $B \leq^{P}_{\kappa} A$ iff some RASMP with base set κ computes B from A.
- $B \leq^{P} A$ iff $B \leq^{P}_{\kappa} A$ for some limit ordinal κ.
- If R is one of the above relations, then
  - R is transitive,
  - R sidesteps the need for an oracle-analogue, and
  - a set x being *R-computable* means $x \; R \; \emptyset$.

Relative Computability Relations

- $B \leq_{\kappa}^{P,s} A$ iff some RASMP with base set κ computes B from A in less than κ time-steps.
- $B \leq_{\kappa}^{P} A$ iff some RASMP with base set κ computes B from A.
- $B \leq^{P} A$ iff $B \leq_{\kappa}^{P} A$ for some limit ordinal κ.
- If R is one of the above relations, then
  - R is transitive,
  - R sidesteps the need for an oracle-analogue, and
  - a set x being *R-computable* means $x \; R \; \emptyset$.

Relative Computability Relations

- $B \leq_{\kappa}^{P,s} A$ iff some RASMP with base set κ computes B from A in less than κ time-steps.
- $B \leq_{\kappa}^{P} A$ iff some RASMP with base set κ computes B from A.
- $B \leq^{P} A$ iff $B \leq_{\kappa}^{P} A$ for some limit ordinal κ.
- If R is one of the above relations, then
  - R is transitive,
  - R sidesteps the need for an oracle-analogue, and
  - a set x being *R-computable* means $x \; R \; \emptyset$.

Degrees of Constructibility

- We show that $B \leq^P A$ iff $B \in L[A]$, where $L[A]$ is Gödel's constructible universe relative to A.
- This means the degrees of relative computability derived from $\leq^P$ are exactly the degrees of constructibility.
- The very restricted (at first glance) programming we allow an RASMPs actually has the full power of transfinite recursion.

Degrees of Constructibility

- We show that $B \leq^P A$ iff $B \in L[A]$, where $L[A]$ is Gödel's constructible universe relative to A.
- This means the degrees of relative computability derived from $\leq^P$ are exactly the degrees of constructibility.
- The very restricted (at first glance) programming we allow an RASMPs actually has the full power of transfinite recursion.

Degrees of Constructibility

- We show that $B \leq^P A$ iff $B \in L[A]$, where $L[A]$ is Gödel's constructible universe relative to A.
- This means the degrees of relative computability derived from $\leq^P$ are exactly the degrees of constructibility.
- The very restricted (at first glance) programming we allow an RASMPs actually has the full power of transfinite recursion.

Degrees of Constructibility

- We show that $B \leq^P A$ iff $B \in L[A]$, where $L[A]$ is Gödel's constructible universe relative to A.
- This means the degrees of relative computability derived from $\leq^P$ are exactly the degrees of constructibility.
- The very restricted (at first glance) programming we allow an RASMPs actually has the full power of transfinite recursion.

The Relations $\leq_{\kappa}^{P,s}$ and $\leq_{\kappa}^{P}$

- An ordinal κ is *admissible* iff $(L_{\kappa}; \in)$ is a model of Kripke-Platek set theory.
- When κ is admissible, $\leq_{\kappa}^{P,s}$ and $\leq_{\kappa}^{P}$ are both defined.
- It can be proven that if κ_1 and κ_2 are admissible and $B \leq_{\kappa_1}^{P,s} A$ (resp. $B \leq_{\kappa_1}^{P} A$), then $B \leq_{\kappa_2}^{P,s} A$ (resp. $B \leq_{\kappa_2}^{P} A$).
- In this case we call $\leq_{\kappa}^{P,s}$ (resp. $\leq_{\kappa}^{P}$) *upward consistent*.

The Relations $\leq_{\kappa}^{P,s}$ and $\leq_{\kappa}^{P}$

- An ordinal κ is *admissible* iff $(L_{\kappa};\in)$ is a model of Kripke-Platek set theory.
- When κ is admissible, $\leq_{\kappa}^{P,s}$ and $\leq_{\kappa}^{P}$ are both defined.
- It can be proven that if κ_1 and κ_2 are admissible and $B \leq_{\kappa_1}^{P,s} A$ (resp. $B \leq_{\kappa_1}^{P} A$), then $B \leq_{\kappa_2}^{P,s} A$ (resp. $B \leq_{\kappa_2}^{P} A$).
- In this case we call $\leq_{\kappa}^{P,s}$ (resp. $\leq_{\kappa}^{P}$) *upward consistent*.

The Relations $\leq_\kappa^{P,s}$ and $\leq_\kappa^P$

- An ordinal κ is *admissible* iff $(L_\kappa; \in)$ is a model of Kripke-Platek set theory.
- When κ is admissible, $\leq_\kappa^{P,s}$ and $\leq_\kappa^P$ are both defined.
- It can be proven that if κ_1 and κ_2 are admissible and $B \leq_{\kappa_1}^{P,s} A$ (resp. $B \leq_{\kappa_1}^{P} A$), then $B \leq_{\kappa_2}^{P,s} A$ (resp. $B \leq_{\kappa_2}^{P} A$).
- In this case we call $\leq_\kappa^{P,s}$ (resp. $\leq_\kappa^P$) *upward consistent*.

The Relations $\leq_\kappa^{P,s}$ and $\leq_\kappa^P$

- An ordinal κ is *admissible* iff $(L_\kappa; \in)$ is a model of Kripke-Platek set theory.
- When κ is admissible, $\leq_\kappa^{P,s}$ and $\leq_\kappa^P$ are both defined.
- It can be proven that if κ_1 and κ_2 are admissible and $B \leq_{\kappa_1}^{P,s} A$ (resp. $B \leq_{\kappa_1}^P A$), then $B \leq_{\kappa_2}^{P,s} A$ (resp. $B \leq_{\kappa_2}^P A$).
- In this case we call $\leq_\kappa^{P,s}$ (resp. $\leq_\kappa^P$) *upward consistent*.

The Relations $\leq_{\kappa}^{P,s}$ and $\leq_{\kappa}^{P}$

- An ordinal κ is *admissible* iff $(L_\kappa; \in)$ is a model of Kripke-Platek set theory.
- When κ is admissible, $\leq_{\kappa}^{P,s}$ and $\leq_{\kappa}^{P}$ are both defined.
- It can be proven that if κ_1 and κ_2 are admissible and $B \leq_{\kappa_1}^{P,s} A$ (resp. $B \leq_{\kappa_1}^{P} A$), then $B \leq_{\kappa_2}^{P,s} A$ (resp. $B \leq_{\kappa_2}^{P} A$).
- In this case we call $\leq_{\kappa}^{P,s}$ (resp. $\leq_{\kappa}^{P}$) *upward consistent*.

Comparisons with Other Relations

In the following table we compare various relative computability relations when they are restricted to subsets of admissible ordinals α.

Property \ Relation	$\leq_\alpha$	$\preceq_\alpha$	$\leq_\alpha^P$	$\leq_\alpha^{P,s}$
Oracle-analogue?	✓	✓	✗	✗
Transitive?	✓	✗	✓	✓
Upward consistent?	✗	✓	✓	✓
Appears in...	α-recursion	α-computability		

Comparisons with Other Relations

In the following table we compare various relative computability relations when they are restricted to subsets of admissible ordinals α.

Property \ Relation	$\leq_\alpha$	$\preceq_\alpha$	$\leq_\alpha^P$	$\leq_\alpha^{P,s}$
Oracle-analogue?	✓	✓	✗	✗
Transitive?	✓	✗	✓	✓
Upward consistent?	✗	✓	✓	✓
Appears in…	α-recursion	α-computability		

Comparisons with Other Relations

In the following table we compare various relative computability relations when they are restricted to subsets of admissible ordinals α.

Property \ Relation	$\leq_\alpha$	$\preceq_\alpha$	$\leq_\alpha^P$	$\leq_\alpha^{P,s}$
Oracle-analogue?	✓	✓	✗	✗
Transitive?	✓	✗	✓	✓
Upward consistent?	✗	✓	✓	✓
Appears in...	α-recursion	α-computability		

Outline

1. New Notions of Generalised (Relative) Computability
   - Computation on Arbitrary Sets
   - Restricting Abstract State Machines
   - The Result
2. Degrees of Small Extensions and Complexity of Local Methods
   - Degrees of Small Extensions
   - Local Method Definitions
   - Results
3. Conclusion

Generators over V

- Degrees of constructibility essentially group sets based on their power as generators over L.
- These degrees can be "computed" in V, based on the previous section.
- Natural generalisation: grouping sets outside V based on their power as generators over V

Generators over V

- Degrees of constructibility essentially group sets based on their power as generators over L.
- These degrees can be “computed” in V, based on the previous section.
- Natural generalisation: grouping sets outside V based on their power as generators over V

Generators over V

- Degrees of constructibility essentially group sets based on their power as generators over L.
- These degrees can be "computed" in V, based on the previous section.
- Natural generalisation: grouping sets outside V based on their power as generators over V

Generators over V

- Degrees of constructibility essentially group sets based on their power as generators over L.
- These degrees can be "computed" in V, based on the previous section.
- Natural generalisation: grouping sets outside V based on their power as generators over V

The Meta-theory

- We do we mean by sets outside V? Isn't V the entire set-theoretic universe?
- Step out of V and treat V as a countable transitive model of ZFC (henceforth denoted CTM).
- Look at those sets that can be adjoined to V to give a nice CTM.

The Meta-theory

- We do we mean by sets outside V? Isn't V the entire set-theoretic universe?
- Step out of V and treat V as a countable transitive model of ZFC (henceforth denoted CTM).
- Look at those sets that can be adjoined to V to give a nice CTM.

The Meta-theory

- We do we mean by sets outside V? Isn't V the entire set-theoretic universe?
- Step out of V and treat V as a countable transitive model of ZFC (henceforth denoted CTM).
- Look at those sets that can be adjoined to V to give a nice CTM.

The Meta-theory

- We do we mean by sets outside V? Isn't V the entire set-theoretic universe?
- Step out of V and treat V as a countable transitive model of ZFC (henceforth denoted CTM).
- Look at those sets that can be adjoined to V to give a nice CTM.

Outer Models

Let U_1 and U_2 be CTMs. U_2 is an outer model of U_1 iff

- $U_1 \subset U_2$, and
- $ORD^{U_1} = ORD^{U_2}$.

Outer Models

Let U_1 and U_2 be CTMs. U_2 is an outer model of U_1 iff

- $U_1 \subset U_2$, and
- $ORD^{U_1} = ORD^{U_2}$.

Outer Models

Let U_1 and U_2 be CTMs. U_2 is an outer model of U_1 iff

- $U_1 \subset U_2$, and
- $ORD^{U_1} = ORD^{U_2}$.

Outer Models

Let U_1 and U_2 be CTMs. U_2 is an outer model of U_1 iff

- $U_1 \subset U_2$, and
- $ORD^{U_1} = ORD^{U_2}$.

Small Extensions

Let U_1 and U_2 be CTMs. U_2 is a small extension of U_1 iff

- U_2 is an outer model of U_1, and
- there is $x \in U_2$ such that U_2 is the smallest outer model of U_1 containing x.

In this case we call x a generator of U_2 over U_1, and denote U_2 as $U_1[x]$.

Small Extensions

Let U_1 and U_2 be CTMs. U_2 is a small extension of U_1 iff

- U_2 is an outer model of U_1, and
- there is $x \in U_2$ such that U_2 is the smallest outer model of U_1 containing x.

In this case we call x a generator of U_2 over U_1, and denote U_2 as $U_1[x]$.

Small Extensions

Let U_1 and U_2 be CTMs. U_2 is a small extension of U_1 iff

- U_2 is an outer model of U_1, and
- there is $x \in U_2$ such that U_2 is the smallest outer model of U_1 containing x.

In this case we call x a generator of U_2 over U_1, and denote U_2 as $U_1[x]$.

Small Extensions

Let U_1 and U_2 be CTMs. U_2 is a small extension of U_1 iff

- U_2 is an outer model of U_1, and
- there is $x \in U_2$ such that U_2 is the smallest outer model of U_1 containing x.

In this case we call x a generator of U_2 over U_1, and denote U_2 as $U_1[x]$.

Small Extensions

Let U_1 and U_2 be CTMs. U_2 is a small extension of U_1 iff

- U_2 is an outer model of U_1, and
- there is $x \in U_2$ such that U_2 is the smallest outer model of U_1 containing x.

In this case we call x a generator of U_2 over U_1, and denote U_2 as $U_1[x]$.

Degrees over V

- In the meta-theory, start with the set $\mathscr{G}$ of all generators of small extensions of V over V.
- Given $x, y \in \mathscr{G}$, $x \leq^S y$ iff $V[x] \subset V[y]$.
- Say $x \equiv^S y$ iff $x \leq^S y$ and $y \leq^S x$.
- $\leq^S$ is transitive so $\equiv^S$ is an equivalence relation.
- Call $(\mathscr{G}/\equiv^S, \leq^S/\equiv^S)$ the degrees of small extensions of V with its standard partial ordering.
- Observe that
$$(\{\text{small extensions of } V\}, \subset) \cong (\mathscr{G}/\equiv^S, \leq^S/\equiv^S),$$
so in many contexts we can interchange "small extension(s)" and "degree(s) of small extensions".

Degrees over V

- In the meta-theory, start with the set $\mathscr{G}$ of all generators of small extensions of V over V.
- Given $x, y \in \mathscr{G}$, $x \leq^S y$ iff $V[x] \subset V[y]$.
- Say $x \equiv^S y$ iff $x \leq^S y$ and $y \leq^S x$.
- $\leq^S$ is transitive so $\equiv^S$ is an equivalence relation.
- Call $(\mathscr{G}/\equiv^S, \leq^S/\equiv^S)$ the degrees of small extensions of V with its standard partial ordering.
- Observe that

$$(\{\text{small extensions of } V\}, \subset) \cong (\mathscr{G}/\equiv^S, \leq^S/\equiv^S),$$

so in many contexts we can interchange "small extension(s)" and "degree(s) of small extensions".

Degrees over V

- In the meta-theory, start with the set $\mathscr{G}$ of all generators of small extensions of V over V.
- Given $x, y \in \mathscr{G}$, $x \leq^S y$ iff $V[x] \subset V[y]$.
- Say $x \equiv^S y$ iff $x \leq^S y$ and $y \leq^S x$.
- $\leq^S$ is transitive so $\equiv^S$ is an equivalence relation.
- Call $(\mathscr{G}/\equiv^S, \leq^S/\equiv^S)$ the degrees of small extensions of V with its standard partial ordering.
- Observe that
$$(\{\text{small extensions of } V\}, \subset) \cong (\mathscr{G}/\equiv^S, \leq^S/\equiv^S),$$
so in many contexts we can interchange "small extension(s)" and "degree(s) of small extensions".

Degrees over V

- In the meta-theory, start with the set $\mathscr{G}$ of all generators of small extensions of V over V.
- Given $x, y \in \mathscr{G}$, $x \leq^S y$ iff $V[x] \subset V[y]$.
- Say $x \equiv^S y$ iff $x \leq^S y$ and $y \leq^S x$.
- $\leq^S$ is transitive so $\equiv^S$ is an equivalence relation.
- Call $(\mathscr{G}/\equiv^S, \leq^S/\equiv^S)$ the degrees of small extensions of V with its standard partial ordering.
- Observe that

$$(\{\text{small extensions of } V\}, \subset) \cong (\mathscr{G}/\equiv^S, \leq^S/\equiv^S),$$

so in many contexts we can interchange "small extension(s)" and "degree(s) of small extensions".

Degrees over V

- In the meta-theory, start with the set $\mathscr{G}$ of all generators of small extensions of V over V.
- Given $x, y \in \mathscr{G}$, $x \leq^S y$ iff $V[x] \subset V[y]$.
- Say $x \equiv^S y$ iff $x \leq^S y$ and $y \leq^S x$.
- $\leq^S$ is transitive so $\equiv^S$ is an equivalence relation.
- Call $(\mathscr{G}/\equiv^S, \leq^S/\equiv^S)$ the degrees of small extensions of V with its standard partial ordering.
- Observe that
$$(\{\text{small extensions of } V\}, \subset) \cong (\mathscr{G}/\equiv^S, \leq^S/\equiv^S),$$
so in many contexts we can interchange "small extension(s)" and "degree(s) of small extensions".

Degrees over V

- In the meta-theory, start with the set $\mathscr{G}$ of all generators of small extensions of V over V.
- Given $x, y \in \mathscr{G}$, $x \leq^S y$ iff $V[x] \subset V[y]$.
- Say $x \equiv^S y$ iff $x \leq^S y$ and $y \leq^S x$.
- $\leq^S$ is transitive so $\equiv^S$ is an equivalence relation.
- Call $(\mathscr{G}/\equiv^S, \leq^S/\equiv^S)$ the degrees of small extensions of V with its standard partial ordering.
- Observe that
$$(\{\text{small extensions of } V\}, \subset) \cong (\mathscr{G}/\equiv^S, \leq^S/\equiv^S),$$
so in many contexts we can interchange "small extension(s)" and "degree(s) of small extensions".

Degrees over V

- In the meta-theory, start with the set $\mathscr{G}$ of all generators of small extensions of V over V.
- Given $x, y \in \mathscr{G}$, $x \leq^S y$ iff $V[x] \subset V[y]$.
- Say $x \equiv^S y$ iff $x \leq^S y$ and $y \leq^S x$.
- $\leq^S$ is transitive so $\equiv^S$ is an equivalence relation.
- Call $(\mathscr{G}/\equiv^S, \leq^S/\equiv^S)$ the degrees of small extensions of V with its standard partial ordering.
- Observe that

$$(\{\text{small extensions of } V\}, \subset) \cong (\mathscr{G}/\equiv^S, \leq^S/\equiv^S),$$

so in many contexts we can interchange "small extension(s)" and "degree(s) of small extensions".

Degrees over V

- In the meta-theory, start with the set $\mathscr{G}$ of all generators of small extensions of V over V.
- Given $x, y \in \mathscr{G}$, $x \leq^S y$ iff $V[x] \subset V[y]$.
- Say $x \equiv^S y$ iff $x \leq^S y$ and $y \leq^S x$.
- $\leq^S$ is transitive so $\equiv^S$ is an equivalence relation.
- Call $(\mathscr{G}/\equiv^S, \leq^S/\equiv^S)$ the degrees of small extensions of V with its standard partial ordering.
- Observe that

$$(\{\text{small extensions of } V\}, \subset) \cong (\mathscr{G}/\equiv^S, \leq^S/\equiv^S),$$

  so in many contexts we can interchange "small extension(s)" and "degree(s) of small extensions".

Outline

1. New Notions of Generalised (Relative) Computability
   - Computation on Arbitrary Sets
   - Restricting Abstract State Machines
   - The Result
2. Degrees of Small Extensions and Complexity of Local Methods
   - Degrees of Small Extensions
   - Local Method Definitions
   - Results
3. Conclusion

Talking about Small Extensions in V

- Often it is useful to refer to small extensions of V within V.
- Such references in general cannot isolate any non-trivial small extension.
- Think of them as a description in V that picks out a subset of $\mathscr{G}/\equiv^S$ when evaluated outside V.
- We want the evaluation of these descriptions to be reasonably absolute, and not be affected by reasonable extensions of the meta-theory.

Talking about Small Extensions in V

- Often it is useful to refer to small extensions of V within V.
- Such references in general cannot isolate any non-trivial small extension.
- Think of them as a description in V that picks out a subset of $\mathscr{G}/\equiv^S$ when evaluated outside V.
- We want the evaluation of these descriptions to be reasonably absolute, and not be affected by reasonable extensions of the meta-theory.

Talking about Small Extensions in V

- Often it is useful to refer to small extensions of V within V.
- Such references in general cannot isolate any non-trivial small extension.
- Think of them as a description in V that picks out a subset of $\mathscr{G}/\equiv^S$ when evaluated outside V.
- We want the evaluation of these descriptions to be reasonably absolute, and not be affected by reasonable extensions of the meta-theory.

Talking about Small Extensions in V

- Often it is useful to refer to small extensions of V within V.
- Such references in general cannot isolate any non-trivial small extension.
- Think of them as a description in V that picks out a subset of $\mathscr{G}/\equiv^S$ when evaluated outside V.
- We want the evaluation of these descriptions to be reasonably absolute, and not be affected by reasonable extensions of the meta-theory.

Talking about Small Extensions in V

- Often it is useful to refer to small extensions of V within V.
- Such references in general cannot isolate any non-trivial small extension.
- Think of them as a description in V that picks out a subset of $\mathscr{G}/\equiv^S$ when evaluated outside V.
- We want the evaluation of these descriptions to be reasonably absolute, and not be affected by reasonable extensions of the meta-theory.

TCIs Again

- A straightforward way to ensure absoluteness is to make evaluation local to the parameters given in the description.
- TCIs and their models are a natural formalisation of this idea.
- A TCI $\mathfrak{T} \in V$ is a description of potential objects in $\mathscr{G}$.
- A model of $\mathfrak{T}$ in the meta-theory is an evaluation of what $\mathfrak{T}$ describes.

TCIs Again

- A straightforward way to ensure absoluteness is to make evaluation local to the parameters given in the description.
- TCIs and their models are a natural formalisation of this idea.
- A TCI $\mathfrak{T} \in V$ is a description of potential objects in $\mathscr{G}$.
- A model of $\mathfrak{T}$ in the meta-theory is an evaluation of what $\mathfrak{T}$ describes.

TCIs Again

- A straightforward way to ensure absoluteness is to make evaluation local to the parameters given in the description.
- TCIs and their models are a natural formalisation of this idea.
- A TCI $\mathfrak{T} \in V$ is a description of potential objects in $\mathscr{G}$.
- A model of $\mathfrak{T}$ in the meta-theory is an evaluation of what $\mathfrak{T}$ describes.

TCIs Again

- A straightforward way to ensure absoluteness is to make evaluation local to the parameters given in the description.
- TCIs and their models are a natural formalisation of this idea.
- A TCI $\mathfrak{T} \in V$ is a description of potential objects in $\mathscr{G}$.
- A model of $\mathfrak{T}$ in the meta-theory is an evaluation of what $\mathfrak{T}$ describes.

TCIs Again

- A straightforward way to ensure absoluteness is to make evaluation local to the parameters given in the description.
- TCIs and their models are a natural formalisation of this idea.
- A TCI $\mathfrak{T} \in V$ is a description of potential objects in $\mathscr{G}$.
- A model of $\mathfrak{T}$ in the meta-theory is an evaluation of what $\mathfrak{T}$ describes.

Local Method Definitions

- A local method definition of V is a definable class of TCIs in V.
- In the meta-theory, define a function Eval^V from the set of TCIs $\mathfrak{T} \in V$ into the set of small extensions of V, such that

$$\mathrm{Eval}^V(\mathfrak{T}) = \{V[M] : M \text{ is a model of } \mathfrak{T}\}.$$

- Each local method definition then picks out a bunch of subsets of $\mathscr{G} / \equiv^S$.

Local Method Definitions

- A local method definition of V is a definable class of TCIs in V.
- In the meta-theory, define a function Eval^V from the set of TCIs $\mathfrak{T} \in V$ into the set of small extensions of V, such that

$$\mathrm{Eval}^V(\mathfrak{T}) = \{V[M] : M \text{ is a model of } \mathfrak{T}\}.$$

- Each local method definition then picks out a bunch of subsets of $\mathscr{G} / \equiv^S$.

Local Method Definitions

- A local method definition of V is a definable class of TCIs in V.
- In the meta-theory, define a function Eval^V from the set of TCIs $\mathfrak{T} \in V$ into the set of small extensions of V, such that

$$\mathrm{Eval}^V(\mathfrak{T}) = \{V[M] : M \text{ is a model of } \mathfrak{T}\}.$$

- Each local method definition then picks out a bunch of subsets of $\mathscr{G} / \equiv^S$.

Local Method Definitions

- A local method definition of V is a definable class of TCIs in V.
- In the meta-theory, define a function Eval^V from the set of TCIs $\mathfrak{T} \in V$ into the set of small extensions of V, such that

$$\mathrm{Eval}^V(\mathfrak{T}) = \{V[M] : M \text{ is a model of } \mathfrak{T}\}.$$

- Each local method definition then picks out a bunch of subsets of $\mathscr{G}/\equiv^S$.

Local Method Definitions

- A local method definition of V is a definable class of TCIs in V.
- In the meta-theory, define a function Eval^V from the set of TCIs $\mathfrak{T} \in V$ into the set of small extensions of V, such that

$$\mathrm{Eval}^V(\mathfrak{T}) = \{V[M] : M \text{ is a model of } \mathfrak{T}\}.$$

- Each local method definition then picks out a bunch of subsets of $\mathscr{G}/\equiv^S$.

Local Method Definitions

- A local method definition of V is a definable class of TCIs in V.
- In the meta-theory, define a function Eval^V from the set of TCIs $\mathfrak{T} \in V$ into the set of small extensions of V, such that

$$\mathrm{Eval}^V(\mathfrak{T}) = \{V[M] : M \text{ is a model of } \mathfrak{T}\}.$$

- Each local method definition then picks out a bunch of subsets of $\mathscr{G}/\equiv^S$.

Comparing Local Methods

- A TCI $\mathfrak{T} \in V$ is consistent iff it has a model in some outer model of V.
- If X and Y are local method definitions of V, $X \leq^M Y$ denotes the statement

  "there is a function $F : X \longrightarrow Y$ definable in V such that $\emptyset \neq \mathrm{Eval}^V(F(\mathfrak{T})) \subset \mathrm{Eval}^V(\mathfrak{T})$ for all consistent $\mathfrak{T} \in X$".

- Intuitively, $X \leq^M Y$ if V can see that Y provides non-trivial refinements to all the descriptions in X.
- $\leq^M$ is transitive, so we can define the equivalence relation $\equiv^M$ in the usual manner.

Comparing Local Methods

- A TCI $\mathfrak{T} \in V$ is consistent iff it has a model in some outer model of V.
- If X and Y are local method definitions of V, $X \leq^M Y$ denotes the statement

  "there is a function $F : X \longrightarrow Y$ definable in V such that $\emptyset \neq \mathrm{Eval}^V(F(\mathfrak{T})) \subset \mathrm{Eval}^V(\mathfrak{T})$ for all consistent $\mathfrak{T} \in X$".
- Intuitively, $X \leq^M Y$ if V can see that Y provides non-trivial refinements to all the descriptions in X.
- $\leq^M$ is transitive, so we can define the equivalence relation $\equiv^M$ in the usual manner.

Comparing Local Methods

- A TCI $\mathfrak{T} \in V$ is consistent iff it has a model in some outer model of V.
- If X and Y are local method definitions of V, $X \leq^M Y$ denotes the statement

  "there is a function $F : X \longrightarrow Y$ definable in V such that $\emptyset \neq \mathrm{Eval}^V(F(\mathfrak{T})) \subset \mathrm{Eval}^V(\mathfrak{T})$ for all consistent $\mathfrak{T} \in X$".

- Intuitively, $X \leq^M Y$ if V can see that Y provides non-trivial refinements to all the descriptions in X.
- $\leq^M$ is transitive, so we can define the equivalence relation $\equiv^M$ in the usual manner.

Comparing Local Methods

- A TCI $\mathfrak{T} \in V$ is consistent iff it has a model in some outer model of V.
- If X and Y are local method definitions of V, $X \leq^M Y$ denotes the statement

  "there is a function $F : X \longrightarrow Y$ definable in V such that $\emptyset \neq \mathrm{Eval}^V(F(\mathfrak{T})) \subset \mathrm{Eval}^V(\mathfrak{T})$ for all consistent $\mathfrak{T} \in X$".
- Intuitively, $X \leq^M Y$ if V can see that Y provides non-trivial refinements to all the descriptions in X.
- $\leq^M$ is transitive, so we can define the equivalence relation $\equiv^M$ in the usual manner.

Comparing Local Methods

- A TCI $\mathfrak{T} \in V$ is consistent iff it has a model in some outer model of V.
- If X and Y are local method definitions of V, $X \leq^M Y$ denotes the statement

  "there is a function $F : X \longrightarrow Y$ definable in V such that $\emptyset \neq \mathrm{Eval}^V(F(\mathfrak{T})) \subset \mathrm{Eval}^V(\mathfrak{T})$ for all consistent $\mathfrak{T} \in X$".
- Intuitively, $X \leq^M Y$ if V can see that Y provides non-trivial refinements to all the descriptions in X.
- $\leq^M$ is transitive, so we can define the equivalence relation $\equiv^M$ in the usual manner.

Comparing Local Methods

- A TCI $\mathfrak{T} \in V$ is consistent iff it has a model in some outer model of V.
- If X and Y are local method definitions of V, $X \leq^M Y$ denotes the statement

  "there is a function $F : X \longrightarrow Y$ definable in V such that $\emptyset \neq \mathrm{Eval}^V(F(\mathfrak{T})) \subset \mathrm{Eval}^V(\mathfrak{T})$ for all consistent $\mathfrak{T} \in X$".

- Intuitively, $X \leq^M Y$ if V can see that Y provides non-trivial refinements to all the descriptions in X.
- $\leq^M$ is transitive, so we can define the equivalence relation $\equiv^M$ in the usual manner.

The Local Method Hierarchy

- We define the complexity of a TCI to be the maximal complexity of sentences in its first-order theory.
- For example, a Σ_n TCI has a first-order theory containing only Σ_n sentences.
- The classes of Σ_n and Π_n TCIs then form a hierarchy under the relation $\leq^M$, called the (small) local method hierarchy.

The Local Method Hierarchy

- We define the complexity of a TCI to be the maximal complexity of sentences in its first-order theory.
- For example, a Σ_n TCI has a first-order theory containing only Σ_n sentences.
- The classes of Σ_n and Π_n TCIs then form a hierarchy under the relation $\leq^M$, called the (small) local method hierarchy.

The Local Method Hierarchy

- We define the complexity of a TCI to be the maximal complexity of sentences in its first-order theory.
- For example, a Σ_n TCI has a first-order theory containing only Σ_n sentences.
- The classes of Σ_n and Π_n TCIs then form a hierarchy under the relation $\leq^M$, called the (small) local method hierarchy.

The Local Method Hierarchy

- We define the complexity of a TCI to be the maximal complexity of sentences in its first-order theory.
- For example, a Σ_n TCI has a first-order theory containing only Σ_n sentences.
- The classes of Σ_n and Π_n TCIs then form a hierarchy under the relation $\leq^M$, called the (small) local method hierarchy.

Set Forcing

- Set forcing is a technique ubiquitous in set theory.
- It can be represented as a local method definition, denoted Fg.
- We want to see if it fits nicely in the local method hierarchy.

Set Forcing

- Set forcing is a technique ubiquitous in set theory.
- It can be represented as a local method definition, denoted Fg.
- We want to see if it fits nicely in the local method hierarchy.

Set Forcing

- Set forcing is a technique ubiquitous in set theory.
- It can be represented as a local method definition, denoted Fg.
- We want to see if it fits nicely in the local method hierarchy.

Set Forcing

- Set forcing is a technique ubiquitous in set theory.
- It can be represented as a local method definition, denoted Fg.
- We want to see if it fits nicely in the local method hierarchy.

Outline

1. New Notions of Generalised (Relative) Computability
   - Computation on Arbitrary Sets
   - Restricting Abstract State Machines
   - The Result
2. Degrees of Small Extensions and Complexity of Local Methods
   - Degrees of Small Extensions
   - Local Method Definitions
   - Results
3. Conclusion

Set Forcing is Π_2

- We show that $\mathrm{Fg} \equiv^M \Pi_2$, where Π_2 is the local method definition containing precisely the Π_2 TCIs.
- This is done by appealing to a set forcing framework developed last year to solve a problem in set theory.

Set Forcing is Π_2

- We show that $\mathrm{Fg} \equiv^M \Pi_2$, where Π_2 is the local method definition containing precisely the Π_2 TCIs.
- This is done by appealing to a set forcing framework developed last year to solve a problem in set theory.

Set Forcing is Π_2

- We show that $\mathrm{Fg} \equiv^M \Pi_2$, where Π_2 is the local method definition containing precisely the Π_2 TCIs.
- This is done by appealing to a set forcing framework developed last year to solve a problem in set theory.

Counting Degrees

- We show that if $\mathfrak{T} \in V$ is a consistent Π_2 TCI, then $\mathrm{Eval}^V(\mathfrak{T})$ is either $\{V\}$ or has size $2^{\aleph_0}$.
- To do this, we formulate an analogue of the Cantor-Bendixson derivative on certain members of Fg, and leverage on the fact that $\Pi_2 \leq^M \mathrm{Fg}$.

Counting Degrees

- We show that if $\mathfrak{T} \in V$ is a consistent Π_2 TCI, then $\mathrm{Eval}^V(\mathfrak{T})$ is either $\{V\}$ or has size $2^{\aleph_0}$.
- To do this, we formulate an analogue of the Cantor-Bendixson derivative on certain members of Fg, and leverage on the fact that $\Pi_2 \leq^M \mathrm{Fg}$.

Counting Degrees

- We show that if $\mathfrak{T} \in V$ is a consistent Π_2 TCI, then $\mathrm{Eval}^V(\mathfrak{T})$ is either $\{V\}$ or has size $2^{\aleph_0}$.
- To do this, we formulate an analogue of the Cantor-Bendixson derivative on certain members of Fg, and leverage on the fact that $\Pi_2 \leq^M \mathrm{Fg}$.

Computability-theoretic Analogues

- A few results linking generic reals — an oft-studied topic in computability theory — with models of countable Π_2 TCIs.
- These results can be proven within V.
- Leverages on the spiritual connection between set forcing in set theory and generic reals in computability theory.
- Requires a closer examination of the set forcing framework used to prove $\mathrm{Fg} \equiv^M \Pi_2$.

Computability-theoretic Analogues

- A few results linking generic reals — an oft-studied topic in computability theory — with models of countable Π_2 TCIs.
- These results can be proven within V.
- Leverages on the spiritual connection between set forcing in set theory and generic reals in computability theory.
- Requires a closer examination of the set forcing framework used to prove $\mathrm{Fg} \equiv^M \Pi_2$.

Computability-theoretic Analogues

- A few results linking generic reals — an oft-studied topic in computability theory — with models of countable Π_2 TCIs.
- These results can be proven within V.
- Leverages on the spiritual connection between set forcing in set theory and generic reals in computability theory.
- Requires a closer examination of the set forcing framework used to prove $\mathrm{Fg} \equiv^M \Pi_2$.

Computability-theoretic Analogues

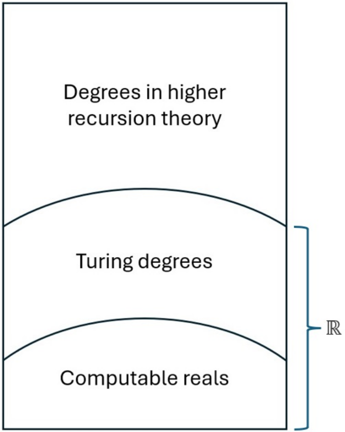
- A few results linking generic reals — an oft-studied topic in computability theory — with models of countable Π_2 TCIs.
- These results can be proven within V.
- Leverages on the spiritual connection between set forcing in set theory and generic reals in computability theory.
- Requires a closer examination of the set forcing framework used to prove $\mathrm{Fg} \equiv^M \Pi_2$.

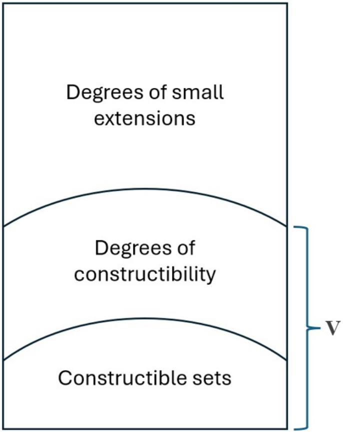
Computability-theoretic Analogues

- A few results linking generic reals — an oft-studied topic in computability theory — with models of countable Π_2 TCIs.
- These results can be proven within V.
- Leverages on the spiritual connection between set forcing in set theory and generic reals in computability theory.
- Requires a closer examination of the set forcing framework used to prove $\mathrm{Fg} \equiv^M \Pi_2$.

Drawing Parallels

Drawing Parallels





Thank You!