CDM

Fundamentals of Computation

Klaus Sutner

Carnegie Mellon University

Fall 2023

Where Are We?

We have an arguably correct notion of computation, based on any of a number of equivalent models: register machines, Turing machines, Herbrand-Gödel equations, μ-recursion, λ-definability.

We can use computability to explain formally what it means to solve a problem. And we have a few examples of interesting problems that are semidecidable but not decidable (beyond just Halting).

The next step is to take a closer look at basic properties of the clone of computable functions.

Enumeration Theorem

Here is yet another way of expressing the fact that there are universal machines.

Theorem

There exists a partial recursive function $\Phi : \mathbb{N} \times \mathbb{N}^n \nrightarrow \mathbb{N}$ such that for every partial recursive function $f : \mathbb{N}^n \nrightarrow \mathbb{N}$ there exists an index $\widehat{f}$ of f such that $f(\boldsymbol{x}) \simeq \Phi(\widehat{f}, \boldsymbol{x})$ for all $\boldsymbol{x}$.

Proof.

One needs to construct a universal device in the corresponding model of computation. For example, for Turing machines the construction was carried out in detail by Turing.

□

Kleene's Notation

We write

$$\{e\}$$

for the eth function, $e \geq 0$, defined by the enumeration theorem. One should think of e as a program (in some suitable language).

Since these functions are partial in general we have to be a bit careful and write

$$\{e\}(x) \simeq y$$

to indicate that $\{e\}$ with input x returns output y after finitely many steps.

More Convergence

To express convergence we also write

$$\{e\}(x)\downarrow$$

if $\{e\}$ on input x terminates and produces some output, and

$$\{e\}(x)\uparrow$$

when the computation fails to terminate.

For example, Kleene equality $\{e\}(x) \simeq \{e'\}(x)$ should be interpreted as:

- either $\{e\}(x)\downarrow$ and $\{e'\}(x)\downarrow$ and the output is the same; or
- $\{e\}(x)\uparrow$ and $\{e'\}(x)\uparrow$.

More Notation

Recall that a model of computation $\mathfrak{M}$ consists of a space of configurations with a one-step relation, plus input and output maps.

The one-step relation is always primitive recursive and different I/O conventions make no difference wrto the computable functions defined by the model.

One really should keep track of arities, so one should write something like

$$\{e\}^{(n)}(\boldsymbol{x}) \simeq y$$

or even $\{e\}^{(n)}_{\mathfrak{M}}(\boldsymbol{x}) \simeq y$. We won't need this level of detail and we won't bother with it.

Robustness

The reason we can (usually) get away with ignoring which particular universal machine we have chosen is the following: suppose we have two different universal machines $\mathcal{U}$ and $\mathcal{U}'$.

Lemma

There is a primitive recursive function f such that

$$\{e\}_{\mathcal{U}}(x) \simeq \{f(e)\}_{\mathcal{U}'}(x)$$

Proof.

By universality, $\mathcal{U}'$ can simulate $\mathcal{U}$, and vice versa. It is straightforward to check (a white lie) that the corresponding transformations are primitive recursive.

□

Cross-Model Simulation

We can even deal with the situation where $\mathcal{U}$ is universal in one model of computation and $\mathcal{U}'$ is universal in another model. Say, $\mathcal{U}$ is a universal Turing machine and $\mathcal{U}'$ is a universal system of Herbrand-Gödel equations.

Then there still is a primitive recursive function f such that

$$\{e\}_{\mathcal{U}}(x) \simeq \{f(e)\}_{\mathcal{U}'}(x)$$

Exercise

Try to get an idea what f would look like for the translation from Turing to Herbrand-Gödel.

Stages of a Computation

Recall the basic idea behind any model of computation: we have a space $\mathcal{C}$ of configurations and a one-step relation $C \left|\frac{1}{\mathfrak{M}}\right. C'$.

But that means that every computation naturally unfolds in stages

$$C_0, C_1, C_2, \ldots, C_\sigma, \ldots, C_t$$

where C_0 is an initial configuration and C_t is a terminal configuration (at least in the finite case).

Abstractly we can write

$$\{e\}_\sigma(x) \simeq y$$

to indicate that $\{e\}$ with input x returns output y after at most σ steps.

Stages are PR

Hence

$$\{e\}(x) \simeq y \iff \exists\, \sigma\ (\{e\}_\sigma(x) \simeq y)$$

Lemma

The relation $R(e, x, y, \sigma) \iff \{e\}_\sigma(x) \simeq y$ is primitive recursive.

Proof.

This is essentially the same argument as for Kleene's T predicate: $C \left|\frac{1}{\mathfrak{M}}\right. C'$ is primitive recursive, whence $C \left|\frac{\sigma}{\mathfrak{M}}\right. C'$ is primitive recursive, uniformly in σ.

□

Dire Warning

But note that the bound σ cannot be computed: there is no total recursive function f such that

$$\{e\}(x) \simeq y \iff \exists\, \sigma < f(e,x)\, \big(\{e\}_\sigma(x) \simeq y\big)$$

Otherwise we could solve the Halting Problem.

However, for practical algorithms (as in 15-451) this is not a problem, we can always predict how long a computation will run. In fact, some elementary bound will work.

Useful Convention

We may safely assume that $\{e\}_\sigma(x)$ always returns a value. To express that the computation has not terminated yet we use the same trick as for bounded search in primitive recursive functions and set

$$\{e\}_\sigma(x) \simeq \sigma$$

By the same token, if the computation already converges in σ steps, all parameters are less than σ:

$$\{e\}_\sigma(x) \simeq y \text{ implies } e, x, y < \sigma$$

Note that the function $\{e\}_\sigma$ is primitive recursive.

We will use the same notation for any computable function f. So

$$f(x) \simeq \lim_{\sigma \to \infty} f_\sigma(x)$$

where the limit is taken in the discrete topology, and may well fail to exist.

This also works the other way around: suppose $f : \mathbb{N}^2 \to \mathbb{N}$ is primitive recursive such that

- $f(x, \sigma) \leq \sigma$, and
- $f(x, \sigma) = y < \sigma$ implies $\forall\, t \geq \sigma\, f(x, t) = y$

Then $f(x) :\simeq \lim f(x, \sigma)$ is computable (partial recursive).

Kleene's S-m-n theorem

Once we have a fixed enumeration, one can compute with indices (think manipulating programs).

Here is a particularly useful, though admittedly quite unspectacular instance of such an index computation: we can fix some of the arguments of a computable function to obtain another computable function.

Theorem

For every $m, n \geq 1$ there is a primitive recursive function S_n^m such that

$$\{S_n^m(e, \boldsymbol{p})\}^{(n)}(\boldsymbol{x}) \simeq \{e\}^{(m+n)}(\boldsymbol{p}, \boldsymbol{x}).$$

Proof.

Klar (pace Landau).

□

Currying

This is just the computability theory version of Currying: we can think of a function f

$$f : A \times B \longrightarrow C$$

as a functional

$$F : A \longrightarrow B \longrightarrow C$$

where $F(a)(b) = f(a, b)$.

In Kleene's version, the emphasis is on the fact that it is easy computationally to get from one to the other.

Swapping Arguments

Note that we can also obtain functions like

$$\lambda x, y.f(5, x, 42, y)$$

since we are always working in a clone.

This may seem a bit nit-picky, but in other frameworks we may have to insist on the existence of transposition operations $T^n_{i,j}(\boldsymbol{x})$ that swap x_i and x_j (and thus provide arbitrary permutations of arguments).

More Index Computation

We claim that there is a primitive recursive function f such that

$$\{f(e, e')\} \simeq \{e\} \circ \{e'\}.$$

Given two devices, we can effectively construct a new device that represents the composition of the given ones.

Or we could use arithmetic to build more complicated functions:

$$\{g(e, e')\} \simeq \{e\} + \{e'\}.$$

And so forth and so on.

Second Recursion Theorem

The next result is utterly amazing: it shows that we solve functional equations of computable functions in mind-numbing generality.

Theorem (Kleene, Second Recursion Theorem, 1938)

Let $F : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ be a partial recursive function. Then there exists an index $e^\star$ such that for all $x \in \mathbb{N}$:

$$\{e^\star\}(\boldsymbol{x}) \simeq F(e^\star, \boldsymbol{x}).$$

Moreover, $e^\star$ can be computed effectively from an index for F.

Think of $e^\star$ as some program, x as input and F as an interpreter running $e^\star$ on x: then the claim is obvious. However, the theorem holds for arbitrary computable F!

No Way

At first glance, this sounds utterly wrong.

What if $F(e, \boldsymbol{x}) = \{e\}(\boldsymbol{x}) + 1$?

Then $\{e^\star\}(\boldsymbol{x}) \simeq \{e^\star\}(\boldsymbol{x}) + 1$.

We already know how to deal with this: any totally undefined function $\{e^\star\}$ works just fine.

In general it requires extra effort to show that a function obtained from the theorem is, say, total. A priori, all we get is a computable function, no more. And, this computable function may be undefined in many places.

Weird Consequences

The theorem has some rather strange consequences:

- Let $F(e, x) \simeq e$. Then $e^\star \simeq \{e^\star\}(x)$.
- Let $F(e, x) \simeq \{x\}(e)$. Then $\{x\}(e^\star) \simeq \{e^\star\}(x)$.

These F are clearly computable, so there is no way around these conclusions–no matter how strange they look.

Quines

The first result, $e^\star \simeq \{e^\star\}(x)$, is the theoretical foundation for quines, programs that print themselves.

The real challenge here is that one needs to deal with the idiosyncrasies (more often: idiocies) of a particular programming language.

Exercise

Write a quine in your favorite programming language.

The Fixed Point Version

Here is yet another version of the recursion theorem that often comes in handy.

Corollary (Rogers, 1967)

Let $f : \mathbb{N} \to \mathbb{N}$ be a recursive function. Then there exists an index $e^\star$ such that

$$\{e^\star\}(\boldsymbol{x}) \simeq \{f(e^\star)\}(\boldsymbol{x}).$$

Proof.

Define $F(e, \boldsymbol{x}) \simeq \{f(e)\}(\boldsymbol{x})$.

□

The Proof

We will prove the second version directly.

Define

$$h(e, \boldsymbol{x}) \simeq F(S_n^1(e, e), \boldsymbol{x})$$

Let $\widehat{h}$ be an index for h and set

$$e^\star := S_n^1(\widehat{h}, \widehat{h})$$

Then

$$\begin{aligned}
\{e^\star\}(\boldsymbol{x}) &\simeq \{S_n^1(\widehat{h}, \widehat{h})\}(\boldsymbol{x}) \\
&\simeq \{\widehat{h}\}(\widehat{h}, \boldsymbol{x}) \\
&\simeq h(\widehat{h}, \boldsymbol{x}) \\
&\simeq F(S_n^1(\widehat{h}, \widehat{h}), \boldsymbol{x}) \\
&\simeq F(e^\star, \boldsymbol{x})
\end{aligned}$$

Aside: Proofs

A proof has (at least) two purposes:

- It establishes the correctness of the claim, beyond a shadow of a doubt.
- It explains why the claim is true.

Whether a proof satsifies these requirements can be a matter of opinion: accepting a proof is a social process, not just a logical one.

Quoi?

The proof of the recursion theorem is one of the most infuriating proofs known to mankind.

Every step is trivial equational reasoning, so arguably the proof is perfect wrto the first requirement.

But the whole argument makes no sense, it explains absolutely nothing.

J. C. Owings called it “barbarically short” and “nearly incapable of rational analysis.”

A Failed Diagonal Argument

Owings also suggested a way to make sense out of it: think of it as a diagonal argument that fails.

In a diagonal argument we have an infinite matrix S over some set A:

$$S : \mathbb{N} \times \mathbb{N} \to A$$

We can think of the rows as infinite sequences over A, so S is a table of infinite sequences.

We have some operation α on A such that the sequence obtained by applying α to the diagonal $\big(\alpha(S(i,i))\big)_{i \geq 0}$ is not in S.

Suppose f is computable and well-behaved on indices: $\{e\} \simeq \{e'\}$ implies $\{f(e)\} \simeq \{f(e')\}$; so f preserves semantics.

Define a matrix S of computable functions by

$$S = \big(\{\{i\}(j)\}\big)_{i,j}$$

Define $\alpha(\{e\}) = \{f(e)\}$.

In this case, the diagonal sequence $(S(i,i))_{i\geq 0}$ as well as its image under α is still a row in S.

The intersection of this row and the diagonal is what we are looking for.

Informal RT

Here is a good intuitive way to think about RT. The typical definition of a computable function Q looks like so:

Q:

input x

some computation

`return` ...

But with the RT we can use an index q for Q inside the definition:

Q:

input x

some computation using q

`return` ...

Roger's Version

This corresponds perfectly to Roger's version of the recursion theorem: the stuff in the box

input x
some computation using q
`return ...`

corresponds to the program transformation f: it modifies program q to some program $f(q)$.

But the modified program turns out to be the same as the original q.

Application: Halting

Assume for the sake of a contradiction that the halting set $K = \{\, e \mid \{e\}(e) \downarrow \}$ is decidable.

Define Q by

```
input x
if char_K(q) ≃ 1        // check q ∈ K
then ↑
else return 0
```

Then $Q(q) \downarrow$ implies $Q(q) \uparrow$, and $Q(q) \uparrow$ implies $Q(q) \downarrow$.

$\uparrow\downarrow\downarrow\uparrow\uparrow\downarrow\downarrow\uparrow\uparrow\downarrow\downarrow\uparrow\uparrow\downarrow\uparrow\downarrow\uparrow\uparrow\uparrow\uparrow\uparrow\uparrow\downarrow\uparrow\downarrow\downarrow\uparrow\uparrow\downarrow\downarrow\uparrow\uparrow\downarrow\uparrow\downarrow\downarrow\uparrow\uparrow\uparrow\downarrow\uparrow\downarrow\uparrow\downarrow\uparrow\uparrow\downarrow\uparrow\uparrow\downarrow$

Application: Ackermann

Define Q by

input x, y
if $x = 0$
then return $y + 1$
elseif $y = 0$
then return $\{q\}(x - 1, 1)$
else return $\{q\}(x - 1, \{q\}(x, y - 1))$

Then $Q = \{q\}$ is none other than the Ackermann function and we have another proof of its computability. Of course, one still has to work to demonstrate totality.

Semi-Characteristic Function

The semi-characteristic function of a relation $A \subseteq \mathbb{N}^k$ is defined by

$$\mathrm{schar}_A(\boldsymbol{x}) \simeq \begin{cases} 0 & \text{if } \boldsymbol{x} \in A, \\ \uparrow & \text{otherwise.} \end{cases}$$

Thus, A is semidecidable iff schar_A is a partial computable function. In fact, A is the domain of definition of schar_A.

Definition

We write W_e for the semidecidable set $\{\, x \mid \{e\}(x) \downarrow \}$.

Thus $(W_e)_e$ is an enumeration of all semidecidable sets, at least if we ignore arities.

Application: Rice's Theorem

Let's say that $P \subseteq \mathbb{N}$ is a non-trivial property of semidecidable sets if

- $W_e = W_{e'}$ implies $e \in P \iff e' \in P$,
- $e_0 \in P$ and $e_1 \notin P$ for some e_0 and e_1.

Examples are "W_e is empty," "W_e is finite," or "W_e is decidable."

Theorem (Rice 1953)

Every non-trivial property of semidecidable sets is undecidable.

Proof

For the sake of a contradiction assume P is decidable.

Define Q by

```
input x
if char_P(q) ≃ 1        // check q ∈ P
then return {e_1}(x)
else return {e_0}(x)
```

But then $q \in P$ implies $Q \simeq \{e_1\}$ and thus $q \notin P$.

On the other hand, $q \notin P$ implies $Q \simeq \{e_0\}$ and thus $q \in P$.

Application: Minimal Machines

All models of computation can be associated with a natural size function. For example, we could define the size of a Turing machine M to be its index $\widehat{M}$ or the number of bits needed to specify its transition function.

Call M minimal if no smaller machine is equivalent to M. Here equivalent means that $\forall\, z\, (M(z) \simeq M'(z))$.

Claim

The set of minimal Turing machines is not semidecidable.

So this is a little stronger than just saying minimality is undecidable.

Proof

For the sake of a contradiction assume minimal machines are semidecidable.

Define Q by

> input x
> enumerate minimal machines M_e until $e > q$
> return $M_e(x)$

But then $Q = \{q\}$ and M_e are equivalent, yet $q < e$, contradicting minimality.

And the First?

Where there is a second, there must be a first. Alas, the First Recursion Theorem is a bit harder to explain, since it uses higher order functionals rather than just functions.

Let us write $\mathcal{P}$ for all partial arithmetic functions (of fixed arity, we will fudge a bit).

We can define a partial order $\sqsubseteq$ on $\mathcal{P}$ by setting $f \sqsubseteq g$ if

$$\forall\, \boldsymbol{x}\, (f(\boldsymbol{x}) \downarrow \text{ implies } f(\boldsymbol{x}) \simeq g(\boldsymbol{x}))$$

Thus f and g agree on the domain of definition of f (g extends f).

Note that this partial order is complete: given an ascending chain (f_i) we can form $\bigsqcup f_i$.

Example

Let $\perp = \emptyset$ be the totally undefined function.

Given any partial function f define the functional F by

$$F(f) = f \cup \{(0,1)\} \cup \{\, (x, x \cdot y) \mid (x-1, y) \in f \,\}$$

Then $f = \bigsqcup F^i(\perp)$ is the factorial function.

As it turns out, all computable functions can be constructed in this manner: by a (easily computable) chain of finite approximations that get closer and closer to the target.

Example

We can also model primitive recursion in this manner. Suppose $f = \mathsf{Prec}[\mathsf{h}, \mathsf{g}]$. Define the functional F by

$$F(f) = f \cup \{\, (0, y, g(y)) \mid y \,\} \cup \{\, (x, y, h(x-1, z, y)) \mid (x-1, y, z) \in f \,\}$$

Then $f = \bigsqcup F^i(\bot)$.

Note that, as written, these approximations are not finite, $F(\bot)$ already contains all of g.

Make sure you understand how F could be adjusted so that all $F^i(\bot)$ are finite. Try addition first.

Good Functionals

Definition

A functional $F : \mathcal{P} \to \mathcal{P}$ is effectively continuous if

- monotonicity: $f \sqsubseteq g$ implies $F(f) \sqsubseteq F(g)$
- continuity: if (f_i) is an ascending sequence in $\mathcal{P}$, then $F(\bigsqcup f_i) = \bigsqcup F(f_i)$
- finite approximations: for some F_{fin} partial recursive $F(\Theta)(\boldsymbol{x}) = F_{\text{fin}}(\widehat{\Theta}, \boldsymbol{x})$ where Θ is a finite function with index $\widehat{\Theta}$.

Compactness

Lemma

Functional F is effectively continuous iff

$$F(f)(\boldsymbol{x}) \simeq y \iff \exists\, \Theta \sqsubseteq f \text{ finite}\,(F(\Theta)(\boldsymbol{x}) \simeq y)$$

In other words, we can already determine the value of $F(f)(\boldsymbol{x})$ by using a suitable finite approximation $\Theta \sqsubseteq f$: no infinite amount of information is needed (say, the values of f on all even numbers).

This comports nicely with any intuitive notion of what it means to effectively compute the functional F.

First Recursion Theorem

Theorem (Kleene, First Recursion Theorem, 1938)

Every effectively continuous functional F has a least fixed point f, a partial recursive function. Moreover, an index for f can be computed effectively from an index for F.

The construction of the "solution" f uses an increasing chain of approximation: Let $f_0 = \bot$ and $f_{i+1} = F(f_i)$. Then

$$f = \bigsqcup f_i = \bigsqcup F^i(\bot)$$

Note that this is essentially call-by-value rather than call-by-name as in the FRT.

Semidecidable Sets and Domains

The definition of a semidecidable set is based on a "semi-algorithm". Alternatively we can use the semi-characteristic function, but note that any other computable function will do as well.

Proposition

A set is semidecidable if, and only if, it is the domain of a partial computable function.

By domain we mean domain of convergence, aka support.

Only convergence matters, the output is irrelevant (unlike with decision algorithms).

Recursively Enumerable Sets

There is another way to look at semidecidable sets: one can generate them in a computable manner.

Definition

$A \subseteq \mathbb{N}$ recursively enumerable (r.e.) if there is a computable function $f : \mathbb{N} \nrightarrow \mathbb{N}$ such that A is the range of f.

Except for $A = \emptyset$ we can choose f to be total. More useful is the following:

Lemma

We may assume without loss of generality that the domain of f is $\mathbb{N}$ or $\{0, 1, \ldots, n-1\}$ for some n, and that f is injective.

Construction for Lemma

Given f, we construct a new function g with the right properties.

We proceed in stages $\sigma \geq 0$. Set $z = 0$.

Stage σ:

Compute $f_\sigma(0), \ldots, f_\sigma(\sigma - 1)$.

If a new value y appears, set $g(z) \simeq y$ and let $z = z + 1$.

Then g is computable, injective, has the same range as f and its support is an initial segment of $\mathbb{N}$, as required.

□

Semidecidable is Recursively Enumerable

Lemma

A set $A \subseteq \mathbb{N}$ is semidecidable iff it is recursively enumerable.

Proof. Suppose A is semidecidable, say $A = \operatorname{dom} f$ for some computable function f.

We construct a computable function g such that $A = \operatorname{rng} g$ in stages:

Stage σ:

Compute $f_\sigma(0), \ldots, f_\sigma(\sigma - 1)$.

If a new value $x < \sigma$ such that $f(x) \downarrow$ appears, set $g(z) \simeq x$ and let $z = z + 1$.

Opposite Direction

Suppose $A = \operatorname{rng} g$.

We construct a computable function f such that $A = \operatorname{dom} f$ in stages:

Stage σ:

Compute $g_\sigma(0), \ldots, g_\sigma(\sigma - 1)$.

If a new value $g(x) \simeq y < \sigma$ appears, set $f(y) \simeq 0$.

□

Convince yourself that this construction can really be handled in a computable manner: there is a primitive recursive function $C(\sigma)$ that computes all the pieces at stage σ.

Approximating Semidecidable Sets

We can approximate semidecidable sets much the way we can approximate computable functions (in fact, it's a bit easier).

Let f be the semi-characteristic function for some semidecidable set W.

Define

$$W_\sigma = \{ x \mid f_\sigma(x) < \sigma \}$$

Note that $W_\sigma \subseteq W$ is a finite set (in fact, it has cardinality at most σ).

Also, W_σ is primitive recursive uniformly in σ.

Lastly, $W = \bigcup W_\sigma$.

We can now give a very careful proof linking decidability and semidecidability.

Lemma

A set A is decidable iff A and $\mathbb{N} - A$ are both semidecidable.

Proof. A decidable means its characteristic function is computable. But then it is easy to see that the semi-characteristic functions of both A and $\mathbb{N} - A$ are also computable.

For the opposite direction let $A = W_e$ and $\mathbb{N} - A = W_{e'}$. Given x let

$$\tau = \min\big(\sigma \mid x \in W_{e,\sigma} \cup W_{e',\sigma} \big)$$

$$f(x) = \begin{cases} 0 & \text{if } x \in W_{e',\tau}, \\ 1 & \text{if } x \in W_{e,\tau}. \end{cases}$$

Then f is the characteristic function of A and clearly is computable. □

Lemma

The collection of semidecidable sets is closed under union and intersection.

Proof.

For union let $A = W_e$ and $B = W_{e'}$. Define

$$f_\sigma(x) \simeq \begin{cases} 0 & \text{if } x \in W_{e,\sigma} \cup W_{e',\sigma}, \\ \uparrow & \text{otherwise.} \end{cases}$$

Then $f = \lim f_\sigma$ is computable and is none other than the semi-characteristic function of $A \cup B$. The argument of intersection is similar. □

In Other Words . . .

And again: this justifies definitions of the form

$$f(x) \simeq \begin{cases} g(x) & \text{if } x \in A, \\ \uparrow & \text{otherwise.} \end{cases}$$

If g is computable and A is semidecidable, then f is also computable.

And again again: we cannot replace the $\uparrow$ in the second case by $h(x)$ unless A is decidable.

Not Even Semidecidable

It follows immediately that the complement $\overline{K}$ of the Halting set is not semidecidable. Let's call such sets co-semidecidable.

So we have to cope with at least three types of problems:

- decidable
- semidecidable
- co-semidecidable

Lemma

A partial function f is computable if, and only if, its graph is recursively enumerable. For total f the graph is decidable.

Proof. Write $F \subseteq \mathbb{N}^2$ for the graph of f.

Suppose f is computable. To semidecide $(x, y) \in F$, we try to compute $f(x)$. If the computation converges, we check that the output is y. i

For the opposite direction, given x, start enumerating F.

If a pair (x, y) appears, output y.

□

Definition

Suppose $R \subseteq \mathbb{N}^{n+1}$. The projection $S \subseteq \mathbb{N}^n$ of R is defined by

$$S(\boldsymbol{x}) \;:\Leftrightarrow\; \exists\, z\, R(z, \boldsymbol{x})$$

Note that S is semidecidable whenever R is decidable: we can search for a witness z.

Lemma

Every semidecidable set is a projection of a decidable set.

This follows immediately from Kleene normal form.