15-150

Principles of Functional Programming

Slides for Lecture 23

Computability

April 17, 2025

Michael Erdmann

https://en.wikipedia.org/wiki/Entscheidungsproblem

The Entscheidungsproblem is a challenge posed by David Hilbert and Wilhelm Ackermann in 1928.

The negative answer to the Entscheidungsproblem was then given by Alonzo Church in 1935–36 (Church's theorem) and independently shortly thereafter by Alan Turing in 1936 (Turing's proof).

Church proved that there is no computable function which decides for two given λ-calculus expressions whether they are equivalent or not. He relied heavily on earlier work by Stephen Kleene.

Turing reduced the question of the existence of a 'general method' which decides whether any given Turing Machine halts or not (the halting problem) to the question of the existence of an 'algorithm' or 'general method' able to solve the Entscheidungsproblem.

The work of both Church and Turing was heavily influenced by Kurt Gödel's earlier work on his incompleteness theorem.

Lessons:

- Decision Questions and Procedures
- Decidability
  - Halting Problem
  - Diagonalization
  - Reduction
- Semi-Decidability
- Some Computability Properties

Language Hierarchy

Class of Languages	Recognizers	Applications
Unrestricted	Turing Machines	General Computation
Context-Sensitive	Linear-bounded automata	Some simple type-checking
Context-Free	Nondeterministic automata with one stack	Syntax checking
Regular	Finite Automata	Tokenization

Some Computational Questions

- Can I win this chess game from this position?
- Does this graph have a cycle consisting of 10 different edges?
- Is this SML expression well-typed?
- Does this SML expression reduce to a value?

Properties

Domain	Problem Instance	Property
chess boards	a specific chess board (arrangement of pieces)	white can win
graphs	a specific graph G	G contains a cycle with 10 edges
SML expressions	a specific `e`	For some `t`, `e : t`.
SML expressions	a specific `e`	For some `v`, `e` $\hookrightarrow$ `v`.

Decision Procedure

Definition

Let P be a property on some domain D.

(we mean a type domain here)

We also assume that for every element **`x`** in D, property P either holds or does not hold, i.e., there is no third possibility.

However, computing whether the property holds can be an issue, as we will see. There exists the third possibility that we will not obtain an answer.

Decision Procedure

Definition

Let P be a property on some domain D.

A *decision procedure for* P is an SML function

`f : D -> bool` such that:

i. **`f(x)`** $\hookrightarrow$ **`true`** if **`P`** holds for instance **`x`**
ii. **`f(x)`** $\hookrightarrow$ **`false`** if **`P`** does not hold for **`x`**
iii. **`f(x)`** returns a value for all **`x`** in D.

Decision Procedure

Definition

Let P be a property on some domain D.

A *decision procedure for* P is an SML function

f : D -> bool such that:

i. **f(x)** $\hookrightarrow$ **true** if **P** holds for instance **x**

ii. **f(x)** $\hookrightarrow$ **false** if **P** does not hold for **x**

iii. **f(x)** returns a value for all **x** in D.

(We state condition (iii) explicitly for emphasis;
we will change it later.)

Decision Procedure

Definition

Let P be a property on some domain D.

A *decision procedure for* P is an SML function

f : D -> bool such that:

i. **f(x)** $\hookrightarrow$ **true** if **P** holds for instance **x**

ii. **f(x)** $\hookrightarrow$ **false** if **P** does not hold for **x**

iii. **f(x)** returns a value for all **x** in D.

- The task of deciding whether property P holds for arbitrary **x** in D is a decision problem.
- When **f** exists as above we say that P is *decidable*.

Example

Domain	Problem Instance	Property P
`regexp * string`	a specific `(r,s)`	`s` ∈ L(`r`)

Property P is decidable.

The regular expression acceptor (with the code that avoids infinite looping for `Star(r)`) provides a decision procedure:

```
fun f (r,s) = accept r s
```

Not all Properties are Decidable

Domain	Problem Instance	Property P
`(int->int) * int`	a specific `(g,x)`	$g(x) \hookrightarrow v$, for some value `v`.

Deciding P is called the Halting Problem.
We will write HALT to mean this P.

Property HALT is not decidable.

Let us prove this fact from the definitions.

Theorem HALT is not decidable.

Proof:

Suppose otherwise, i.e., suppose there exists

`H : (int->int)*int -> bool` such that

i. **`H(g,x)`** $\hookrightarrow$ **`true`** if **`g(x)`** is valuable
ii. **`H(g,x)`** $\hookrightarrow$ **`false`** if **`g(x)`** is not valuable
iii. **`H(g,x)`** returns a value for all **`(g,x)`**.

Now define:

```
fun loop () = loop ()

fun diag (x:int):int =
        if H(diag,x) then loop () else 0
```

We will see that this reasoning leads to a contradiction.
So **`H`** cannot exist, establishing the theorem.

Consider now `H(diag,0)`.

By property (iii) of **H**, this expression reduces to either **true** or **false**. Let us examine each possibility.

<u>`H(diag,0)` $\hookrightarrow$ `true`</u>   Let's evaluate `diag(0)`:

`diag(0)` $\Longrightarrow$ `if H(diag,0) then loop() else 0`
$\Longrightarrow$ `loop()`

So **H** says `diag(0)` is valuable, but it actually loops forever.

<u>`H(diag,0)` $\hookrightarrow$ `false`</u>   Again, let's evaluate:

`diag(0)` $\Longrightarrow$ `if H(diag,0) then loop() else 0`
$\Longrightarrow$ `0`

So **H** says `diag(0)` is not valuable, but it actually reduces to `0`.

For both possibilities we obtain a contradiction.

QED

Proof Techniques

- The previous proof technique is known as a *diagonalization argument*. It sets up an adversary who does the opposite of what is expected (very similar to Cantor's proof that the reals are uncountable).

- Another common proof technique is a *reduction argument* (to be discussed next).

Reduction Argument

Let **P** and **Q** be two properties.

We write $\mathtt{f}_{P}$ to mean a decision procedure for **P** and $\mathtt{f}_{Q}$ to mean a decision procedure for **Q**.

We say that **P** is *reducible to* **Q** if, given $\mathtt{f}_{Q}$, one could implement $\mathtt{f}_{P}$ by calling $\mathtt{f}_{Q}$ on the result of transforming the arguments passed to $\mathtt{f}_{P}$ (intuitively, if $\mathtt{f}_{P} = \mathtt{f}_{Q} \circ \mathtt{i}$ for some total function $\mathtt{i}$).

OBSERVE:

provides a proof roadmap

If **P** is reducible to **Q** and if $\mathtt{f}_{P}$ is known not to exist, then $\mathtt{f}_{Q}$ cannot exist.

We might think that HALT is undecidable merely because there are infinitely many possible arguments **`x`**, so let's look at a variant:

Domain	Problem Instance	Property P
`int -> int`	a specific **`g`**	**`g(0)`** $\hookrightarrow$ **`v`**, for some value **`v`**.

We will write $HALT_0$ to mean this P.

Property $HALT_0$ is also not decidable.

Let us prove this fact by reducing HALT to $HALT_0$.

Theorem $HALT_0$ is not decidable.

Proof: By reduction, reducing HALT to $HALT_0$.

Let **`Z`** mean a decision procedure for $HALT_0$.
Let **`H`** mean a decision procedure for HALT.

We proved earlier than **`H`** does not exist.
We will show that if **`Z`** existed, then we could define **`H`**.
Consequently, **`Z`** cannot exist.

```
fun H (g:int->int, x:int) : bool =
        Z ( fn (y:int) => g x )
```

Observe that **`H`** is total since **`Z`** is. Moreover,
`H(g,x)` returns **`true`** iff **`Z(fn ...)`** returns **`true`** iff
`(fn y => g x)(0)` is valuable iff **`g(x)`** is valuable.
So **`H`** would indeed be a decision procedure for HALT. QED

Comment

Be careful about the direction of the reduction.

For instance, one could also define

```
fun Z (g : int -> int) : bool = H(g, 0)
```

That would be a reduction of $HALT_0$ to HALT. It would **not** help us prove that $HALT_0$ is undecidable.

Comment

Be careful about the direction of the reduction.

For instance, one could also define

```
fun Z (g : int -> int) : bool = H(g, 0)
```

That would be a reduction of HALT$_0$ to HALT. It would **not** help us prove that HALT$_0$ is undecidable.

However, the two reductions together tell us that HALT and HALT$_0$ are "equivalently undecidable".

A Computability Hierarchy

The phrase "equivalently undecidable" suggests degrees of undecidability.

Let us explore that idea a little.

Recall: Decision Procedure

Definition

Let P be a property on some domain D.

A *decision procedure for* P is an SML function

`f : D -> bool` such that:

i. **`f(x)`** $\hookrightarrow$ **`true`** if **`P`** holds for instance **`x`**
ii. **`f(x)`** $\hookrightarrow$ **`false`** if **`P`** does not hold for **`x`**
iii. **`f(x)`** returns a value for all **`x`** in D.

We will now remove condition (iii) by changing condition (ii).

Semi-Decision Procedure

Another definition

Let P be a property on some domain D.

A *semi-decision procedure for* P is an SML function

`f` **`:`** D **`->`** **`bool`** such that:

i. **`f(x)`** $\hookrightarrow$ **`true`** if **`P`** holds for instance **`x`**

ii. **`f(x)`** $\hookrightarrow$ **`false`** **OR** **`f(x)`** diverges
if **`P`** does not hold for **`x`**.

In other words, **`f`** must return **`true`** for instances **`x`** that satisfy P, but can either return **`false`** or diverge for instances that do not satisfy P.

("diverge" means "does not return a value")

Semi-Decision Procedure

Another definition

Let P be a property on some domain D.

A *semi-decision procedure for* P is an SML function

`f : D -> bool` such that:

i. **`f(x)`** $\hookrightarrow$ **`true`** if **`P`** holds for instance **`x`**

ii. **`f(x)`** $\hookrightarrow$ **`false`** **OR** **`f(x)`** diverges
if **`P`** does not hold for **`x`**.

- When **`f`** exists as above we say that P is *semi-decidable*.

Theorem HALT is semi-decidable.

Proof:

Here is a semi-decision procedure for HALT:

```
fun S (g:int->int, x:int) : bool = (g x; true)
```

QED

Theorem HALT is semi-decidable.

Proof:

Here is a semi-decision procedure for HALT:

```
fun S (g:int->int, x:int) : bool = (g x; true)
```

QED

Theorem $HALT_0$ is semi-decidable.

Proof:

```
fun S0 (g:int->int) : bool = (g 0; true)
```

QED

Co-Semi-Decidability

Yet another definition

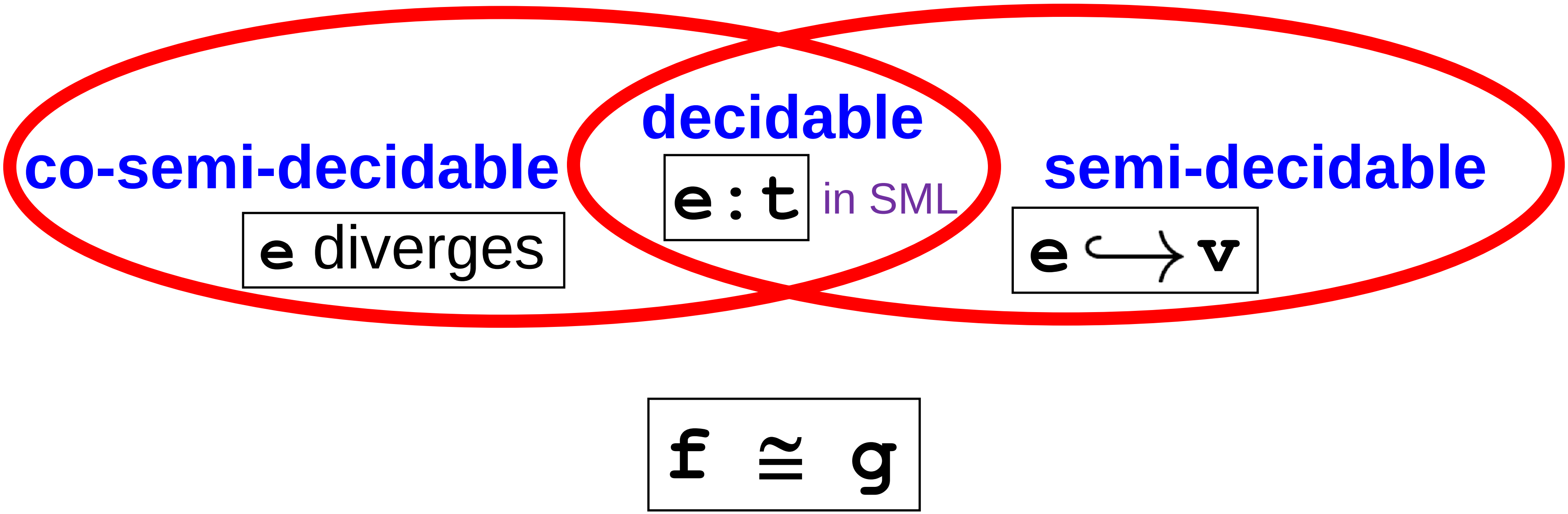
Let P be a property on some domain D.

We say that P is *co-semi-decidable* if ¬P is semi-decidable.

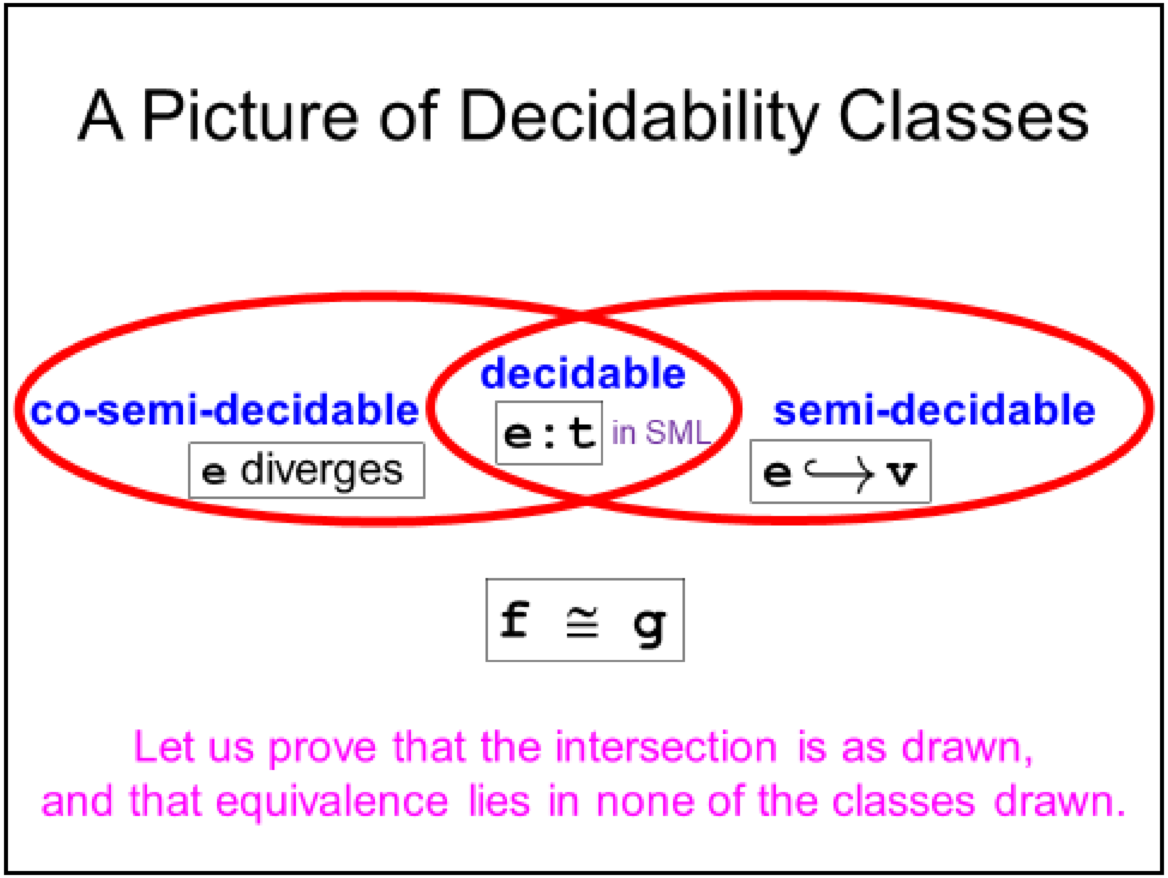
(¬P means the Boolean negation of P.)

For example, the property "**`g(0)`** diverges" is co-semi-decidable since $HALT_0$ is semi-decidable.

A Picture of Decidability Classes



Let us prove that the intersection is as drawn,
and that equivalence lies in none of the classes drawn.



We will prove some other results along the way. Doing so will help achieve our goal, as well as build some intuition about the "calculus of undecidability".

Theorem Let P be a property on some domain D.

P is decidable if and only if ¬P is decidable.

Proof:

Let $\mathbf{f}_P$ be a decision procedure for P.

We can define a decision procedure $\mathbf{g}_{\neg P}$ for ¬P:

```
fun g¬P(x) = not(fP(x))
```

$\mathbf{g}_{\neg P}$ is total since $\mathbf{f}_P$ is, and decides ¬P correctly since $\mathbf{f}_P$ decides P correctly.

(The other direction of the "iff" is similar.)

QED

Theorem Let P be a property on some domain D.

If P is both semi-decidable and co-semi-decidable, then P is in fact decidable.

Proof:

Let f_P be a semi-decision procedure for P and let $g_{\neg P}$ be a semi-decision procedure for ¬P.

We define a decision procedure **h : D -> bool** for P:

For a given problem instance **x** in D, **h** interleaves evaluation of f_P**(x)** and $g_{\neg P}$**(x)**.

At least one of these expressions is valuable.

If f_P**(x)** returns a value before $g_{\neg P}$**(x)** does, then **h(x)==>**f_P**(x)**. Otherwise, **h(x)==>not(**$g_{\neg P}$**(x))**.

QED

Theorem $HALT_0$ is not co-semi-decidable.

Proof:

We saw earlier that $HALT_0$ is semi-decidable.

If $HALT_0$ were also co-semi-decidable, then the previous theorem would imply that $HALT_0$ is decidable, which we proved earlier is not the case.

QED

Let us now consider function equivalence.
We assume the pure subset of SML
that does not include mutation or exceptions.

Domain: `(int -> int) * (int -> int)`

Problem Instance: a specific pair `(f,g)`

Property: $\mathbf{f} \cong \mathbf{g}$

We will write EQUIV to mean this property.

Theorem EQUIV is neither semi-decidable nor co-semi-decidable.

Proof: (Reduction arguments make sense for semi-decidability.)

1. Suppose **Eq** is a semi-decision procedure for EQUIV. Then **s** below would be a semi-decision procedure for ¬$HALT_0$, contradicting $HALT_0$ being not co-semi-decidable:

```
fun s (h:int->int):bool =
      Eq (fn (y:int) => (h 0; y),
          fn (y:int) => loop ())
```

2. If **notEq** is a semi-decision procedure for ¬EQUIV, then **s′** would be a semi-decision procedure for ¬$HALT_0$:

```
fun s′ (h:int->int):bool =
      notEq (fn (y:int) => (h 0; y),
             fn (y:int) => y)
```

QED

One more comment

The (un)decidability properties we discussed today do not depend on our working with SML functions.
The same properties hold if we were to examine the abstract syntax trees of written code or if we were to work in a different programming language or at the assembly level or even at the transistor level of a computer.

That is all.

Please have a good weekend.

See you Tuesday.

We will discuss automated game playing.