Rice's theorem

1 Undecidability by Reduction

See also the video lecture recording: Undecidability by Reduction on Canvas.

Let P and Q be problems. To *reduce* problem P to problem Q means to give a way of solving P using a black box that solves Q. For example, I saw a recipe for profiteroles that said "take some pastry balls" and "take some chocolate sauce". The author reduced the problem of making profiteroles to the problems of making pastry balls and making chocolate sauce.

Suppose we've reduced problem P to problem Q.

- If Q is decidable, then P is decidable
- If P is undecidable, then Q is undecidable.

Now that we know the halting property is undecidable, we can deduce the undecidability of many other properties. Here's an example. A program

```
void f () {
  ...
}
```

is *orange* when it both halts and contains (in the body code) more occurrences of "a" than "b". Thus

```
int a = 3;
return;
```

is orange, but

```
int b = 3;
return;
```

is not, and

```
int a = 3;
while (true) {}
```

is not. Orangeness is undecidable: to prove this fact, we reduce the halting problem to the orangeness problem. For any program C of type `void`, let $F(C)$ be the same code as C with an extra comment line at the end consisting of just enough occurrences of "a" so that there are more occurrences of "a" than "b". Then C halts iff $F(C)$ is orange. So if orangeness were decidable, then the halting property would be too.

2 Semantic and non-semantic properties

See also the video lecture recording: Introducing Rice's theorem on Canvas.

As you may have noticed, orangeness is a rather strange property. Just look at the examples above: the program

```
int a = 3;
return;
```

is orange, but

```
int b = 3;
return;
```

is not. Yet these programs have exactly the same *semantics*, i.e. the same behaviour observable by an external user. While orangeness is a property of code, it isn't a property of behaviour.

When you want to test that code is good or bad, you typically aren't interested in properties like orangeness. The properties you are interested in are properties of behaviour. They are called *semantic properties*. For example:

- Does this software print only polite words?
- Does this software provide good advice to all users, and excellent advice to premium users?
- Does this software, when supplied with two positive integers, always return the highest common factor?

All these questions concern only the behaviour of the program and nothing else. They are semantic properties. It would be great if we could check them automatically.

Well, we can't. *Rice's Theorem* says that (except in two cases, which I'll come to later), every semantic property is undecidable.

Recap To show a property (such as orangeness) is not semantic, give two programs that have the same semantics (behaviour), but one of them has the property and the other one doesn't. In the absence of such a pair, the property is semantic.

Exercise A program

```
void f () {
  ...
}
```

is *red* when it prints a friendly greeting, and the code comments include a poem. Show that redness is not semantic.

3 The two exceptions

Look at the following example:

- Does this software (of type `nat`) return a number that is greater than itself?

This is a semantic property: it concerns the behaviour of a program. But the answer is always No. So this property is decidable: we can test for it by a one-line program that just returns False.

Now look at the following example:

- Does this software (of type `nat`) either hang, or return a number that is equal to itself?

This is a semantic property: it concerns the behaviour of a program. But the answer is always Yes. So this property is decidable: we can test for it by a one-line program that just returns True.

Now look at the following example:

- Does this software print only polite words?

This is a semantic property: it concerns the behaviour of a program. There's some program that satisfies it (i.e. prints only polite words), and some program that doesn't. Rice's Theorem tells us that it is undecidable:

To summarize, there are three kinds of semantic property:

- A property that never holds. This is decidable.
- A property that always holds. This is decidable.
- A property that holds in some case and fails to hold in some case. This is undecidable.

Rice's Theorem *Any semantic property of code that holds in some case and fails to hold in some case is undecidable.*

4 Proving Rice's Theorem

See also the video lecture recording: Proof of Rice's theorem on Canvas.

To prove Rice's Theorem, we use the same method that we've seen before: reduction of the Halting Problem. There are actually two kinds of situation. Look at these examples:

1. Consider the following example:

   - A method `void f ()` is *polite* when it prints only polite words.

   Note that the code

   ```
   while (true) {}
   ```

   is polite but the code

```
System.out.println("You #$@&%*!");
```

is not. So given any code P of type `void`, we see that P halts iff the code

P

```
System.out.println("You #$@&%*!");
```

is not polite.

2. Now look at this example:

   - The software `void f (boolean premium)` is *helpful* when it prints helpful advice to all users, and excellent advice to premium users.

   Note that the code

```
while (true) {}
```

is not helpful but the code

```
if premium {
  System.out.println("Unpack all definitions before "
      + "attempting a proof");
} else {
  System.out.println("Check your answer before "
      + "handing it in.");
}
```

is helpful. So given any code P of type `void`, we see that P halts iff the code

P

```
if premium {
  System.out.println("Unpack all definitions before "
     + "attempting a proof");
} else {
  System.out.println("Check your answer before "
     + "handing it in.");
}
```

is helpful.

Now in general: given a semantic property $\mathcal{R}$ that holds in some case and fails to hold in some case, we ask whether the code `while (true)` that just hangs has the property.

- If it does, then there must be some other piece of code M that doesn't. Now for any code P of type `void`, let $F(P)$ be the code

  $$\begin{array}{c} P \\ M \end{array}$$

  If P halts, then $F(P)$ has the same semantics as M, so $F(P)$ doesn't satisfy $\mathcal{R}$, just like M doesn't. If P hangs, then $F(P)$ has the same semantics as `while (true)` , so $F(P)$ satisfies $\mathcal{R}$, just like `while (true)` . To summarize, P halts iff $F(P)$ does not have the property $\mathcal{R}$. So we have reduced the halting problem to $\mathcal{R}$. So $\mathcal{R}$ is undecidable.

- If it doesn't, then we apply the same argument the other way round.

5 Decidability in Practice

See also the video lecture recording: Decidability in Practice on Canvas.

Let us look at how undecidability questions crop up in practice.

Verification. Because it is so common to write bugs when programming, some people have developed a more formal approach to software development. Someone first writes a *specification*, which describes properties of the code. (This may be written in a specification language such as Z.) Then the program is written. Finally we want to check the program automatically to see if it meets the specification. However, this is undecidable. In some cases, the verifier will say "Yes, the code meets its spec." In other cases it will find bugs. But necessarily there will be cases where the verifier cannot establish whether the program meets its spec or not.

Type reconstruction. In some programming languages, such as Haskell and ML, variables don't need to be declared, because the system works out the correct type automatically. However, while this works for these languages, it would not work for a stronger language than Haskell, because general type inference is undecidable. Knowing this, the designers of Haskell deliberately constrained the language so as to have automatic type inference for it.

Mobile code and viruses. If you allow code from an external source to run on your machine, then you run the risk of that code performing destructive actions on your data. It would be extremely useful if we could test code automatically for potential malicious behaviour. Again, this is impossible.

There are two partial solutions to this problem.

- Virus checking software detects some viruses but it is incomplete. No matter how sophisticated it is, there will always be damaging code that it is not able to recognize.
- Alternatively, we can be overly conservative: run code in a *sandbox* that guarantees that it can't access any important data, except in cases where we know that the access is safe. But there will always be some safe kinds of execution that we disallow.

Logic. Proving theorems is hard! Wouldn't it be great if we could determine automatically which statements are provable? But even in very simple kinds of logic (e.g. predicate logic), this is undecidable. And the reason is familiar: reduce the Halting Problem to provability of sentences in predicate logic.