Decidability III

David L. Dill
Department of Computer Science
Stanford University

Announcements
Homework due now.
Next homework will be posted on the web.

Outline
1 Decidability
   1. Mapping reduction
   2. Nondeterministic Programs

Section
Decidability
A Simple Function

This is just LURec, but with an easier-to-remember name.

Given arguments \( f \) (a string representing a Javascript function) and \( w \) (a string), it returns “true” if \( f(w) \) returns true.

```javascript
function accepts(f, w) {
    // This constructs a function call f('w'),
    // with the actual parameters substituted for
    // f and w.
    // Parenthesizing f prevents some parsing errors.
    var fcall = "(" + f + ")" + "(" + w + ")";
    // eval runs fcall in Javascript and returns the
    // result. If fcall loops, so will eval(fcall).
    return eval(fcall);
}
```

Then we can define LURec:
```javascript
function LURec(f, w) { return accepts(f, w); }
```

Computable Functions

Definition (Computable Function)

A **computable function** is a function \( f : \Sigma^*_1 \rightarrow \Sigma^*_2 \) by a program that always halts with the correct output.

Definition (Mapping Reduction)

A **mapping reduction** from \( A \) to \( B \) is a computable function \( f : \Sigma^*_1 \rightarrow \Sigma^*_2 \), where \( w \in A \) iff \( f(w) \in B \) for all \( w \in \Sigma^*_1 \).

If there is a mapping reduction from \( A \) to \( B \), we write \( A \leq_m B \).

Intuition: \( A \leq_m B \) means “\( A \) is no harder than \( B \)” (or “\( B \) is at least as hard as \( A \)”).

Mapping Reduction and Negative Results

**Theorem**

If \( A \subseteq \Sigma^*_1 \) and \( B \subseteq \Sigma^*_2 \) and \( A \leq_m B \), then

- if \( A \) is not co-RE, then \( B \) is not co-RE.
- if \( A \) is not RE, then \( B \) is not RE.
- if \( A \) is not decidable, then \( B \) is not decidable.

Mapping reduction allows us to prove that some languages are not RE and not co-RE.
Advice for Working with Mapping Reductions

Ask yourself

- What is the input to the mapping function?
- What is the output of the mapping function?
- What is the property \( w \in A \text{ iff } f(w) \in B \) for our specific problems \( A \) and \( B \)?

Mapping functions often require functions to “build” strings for new functions.

---

Let \( L_\epsilon = \{ h \mid \epsilon \in L(h) \} \) (programs that accept the empty string).

\( L_\epsilon \) is obviously RE. This function recognizes it (\( \epsilon \) in Javascript is ""):

```javascript
function LepsRec (h) { return accepts(h, ""); }
```

**Guess:** \( L_\epsilon \) is probably not decidable, because we can’t tell if a function is looping on \( \epsilon \) or just taking a long time.

---

**Theorem**

\( L_\epsilon \) is not co-RE.

**proof.** We prove \( L_U \leq_m L_\epsilon \) by giving a mapping reduction. Since \( L_U \) is not co-RE, this will show that \( L_\epsilon \) is not co-RE.

What are the inputs and outputs of the mapping reduction?

**Input:** A problem instance of \( L_U \): \( \langle g, w \rangle \) where \( g \) is a Boolean function on strings.

**Output:** A problem instance of \( L_\epsilon \): \( h \) where \( h \) is a Boolean function on strings.

So, the mapping function has to take a function as an argument and it returns a function string.
Mapping Reduction for $L_U$ to $L_\epsilon$

The mapping function $f$ builds a new function out of $g$ and $w$:

```javascript
function f(g, w)
{
    return "function h(x) {return (" + g + "(" + w + ")});";
}
```

$f(g', 'w')$ returns

```javascript
function h(x) { return (g('w'));}
```

To prove that $f$ is a mapping function, we need to prove that $g$ accepts $w$ iff $h$ accepts $\epsilon$.

**Note:** $h$ ignores the argument $x$ and runs $g$ on $w$.

Proof that $L_\epsilon$ is not co-RE, cont.

Calling the mapping function on $g$ and $w$ returns:

```javascript
function h(x) { return g('w');}
```

- If $g(w) = \text{true}$, then $h$ returns true, so $h$ accepts $\epsilon$.
- If $g(w) = \text{false}$, then $h$ returns false, so $h$ does not accept $\epsilon$.
- If $g(w)$ loops, then $h$ loops, so $h$ does not accept $\epsilon$.

Hence, $h$ accepts $\epsilon$ iff $g$ accepts $w$.

We have a mapping reduction from $L_U$ to $L_\epsilon$, so $L_U \leq_m L_\epsilon$. $L_U$ is not co-RE, so $L_\epsilon$ is not co-RE. 

Testing in Javascript

Before resuming the proof, let's test $f$:

```javascript
> var h = f("function (a) { return(s[0] === 'f'); }", "foo");
> accepts(h, "")
true

> var h = f("function (a) { return(s[0] === 'g'); }", "foo");
> accepts(h, "")
false

> var h = f("function (a) {while (true);" +
    "return(s[0] === 'f');}"", "foo");
> accepts(h, "")
(nothing)
```

Inside Mapping Reduction

We proved that $L_\epsilon$ is not co-RE, i.e. $\overline{L_\epsilon}$ is not RE.

Suppose we had a function $\text{LepsBarRec}(h)$ that halts whenever $h$ does not accept $\epsilon$.

Then we could write a function that takes $g$ and $w$ as arguments, and halts whenever $g$ does not accept $w$

```javascript
function LUBarRec(g, w)
{
    var h = f(g, w);
    return LepsBarRec(h);
}
```

but $L_U$ is not co-RE, so $\overline{L_U}$ is not RE.

$h$ is Just Data

The function $h$ is never actually run! It is simply an object created to "trick" $\text{LepsBarRec}$ into solving $L_U$. 

The Emptiness Problem is not RE

Let $L_{\emptyset} = \{ g \mid L(g) = \emptyset \}$.

**Theorem**

$L_{\emptyset}$ is not RE.

**proof.** We show that $L_U \leq_m L_{\emptyset}$. Since $L_U$ is not RE, this shows that $L_{\emptyset}$ is not RE.

First, let’s think about the argument and return value of the mapping function $f$:

**Argument:** an instance of $L_U$: $\langle g, w \rangle$.

**Return value:** $h$, a string representing a Boolean function.

We want $g$ not to accept $w$ iff $L(h) = \emptyset$.

I.e. $g$ accepts $w$ iff $L(h) \neq \emptyset$.

We can use the same mapping function as in the previous proof:

```
function f(g, w)
{
    return "function h(x) {return " + g + "(" + w + ")};";
}
```

$f(\langle g', w' \rangle)$ returns the function:

```
function h(x) { return g('w');}
```

If $g$ accepts $w$, then $h$ accepts all $x$, so $L(h) \neq \emptyset$.

If $g$ rejects $w$, then $h$ rejects all $x$, so $L(h) = \emptyset$.

If $g$ loops on $w$, then $h$ loops on all $x$, so $L(h) = \emptyset$.

So, $g$ not to accept $w$ iff $L(h) = \emptyset$ as required.

Hence, $L_U \leq_m L_{\emptyset}$, so $L_{\emptyset}$ is not in RE.  □.

---

A problem that is neither RE nor co-RE

**Theorem**

$L_{\text{EQ}} = \{ \langle g, h \rangle \mid L(g) = L(h) \}$ is neither RE nor co-RE.

$L_{\text{EQ}}$ is a really hard language.

The only method we have that could prove this is mapping reduction.

We do two reductions, one to show that $L_{\text{EQ}}$ is not RE and one to show $\overline{L_{\text{EQ}}}$ is not RE.

In the proof, we need to be able to construct some simple programs:

Given $\langle g, w \rangle$, we need to do the following:

Construct $g_1$ where $L(g_1) = \emptyset$. This is easy:

```
function g1(x) { return false; }
```

Construct $g_2$ where $L(g_2) = \Sigma^*$. Also, easy:

```
function g2(x) { return true; }
```

Note that both functions ignore their argument $x$. 

---
**L\(_{EQ}\) is neither RE nor co-RE, cont.**

We also need to be able to construct \(g_3\) where

\[
\begin{align*}
L(g_3) &= \Sigma^* \quad \text{if } w \in L(g) \\
L(g_3) &= \emptyset \quad \text{if } w \notin L(g)
\end{align*}
\]

It’s almost as easy:

function \(g_3\) (\(x\)) { return \(g(w)\); }

Note that this function also ignores its argument.

(In the next slide, \(f_1\) will return \([g_1, g_3]\) instead of combining them into a single string, to keep the code simple.)

---

**L\(_{EQ}\) is neither RE nor co-RE, cont.**

**proof.** We prove \(L_{EQ}\) is not RE by showing \(\overline{L_U} \leq_m L_{EQ}\) and prove \(L_{EQ}\) is not co-RE by showing \(L_U \leq_m L_{EQ}\).

The mapping function for \(L_U \leq_m L_{EQ}\) is:

function \(f_1\)(\(g, w\))
{
    var \(g_1 = \text{"function } g_1(x) \{ \text{return false; \};\};"\);
    var \(g_3 = \text{"function } g_3(x) \{ \text{return } g(w); \};\};"
    return \([g_1, g_3]\);
}

We need to show \((g, w) \notin L_U\) iff \(L(g_1) = L(g_3)\).

\(L(g_1) = \emptyset\) in all cases. If \((g, w) \notin L_U\) then \(g(w)\) loops or returns false, so \(L(g_3) = L(g_1)\).

Otherwise, \(g(w) = \text{true}\), \(L(g_3) = \Sigma^* \neq L(g_1)\). So \((g, w) \in \overline{L_U}\) iff \(L(g_1) \neq L(g_3)\) and \(L_U \leq_m L_{EQ}\).

---

**L\(_{EQ}\) is neither RE nor co-RE proof, cont.**

To show that \(L_U \leq_m L_{EQ}\), we define:

function \(f_2\)(\(g, w\))
{
    var \(g_2 = \text{"function } g_2(x) \{ \text{return true; \};\};\);
    var \(g_3 = \text{"function } g_3(x) \{ \text{return } g(w); \};\};"
    return \([g_2, g_3]\);
}

\(L(g_2)\) is always \(\Sigma^*\). If \(g(w) = \text{true}\) then \(L(g_2) = \Sigma^* = L(g_3)\). Otherwise, \(L(g_3) = \emptyset\), so \(L(g_3) \neq L(g_2)\). So \((g, w) \in L_U\) iff \(L(g_1) = L(g_3)\), as required. □

---

**Subsection**

**Nondeterministic Programs**
Nondeterminism in Programs

Real-world computers don’t have nondeterminism, but it’s important in models of computation.

We could add nondeterminism to Javascript by adding a `guess` operation that chooses a value from a finite set.

The `guess` operation is magic. **If there is any choice that can cause the function to halt and return true, even at a distant future time, it will return that choice.**

`guess S` Chooses a value for `p` from an array of values `S`.

Acceptance by a Nondeterministic Program

**Definition**
A nondeterministic program accepts its argument if some sequence of choices results returning true.

**Definition**
A nondeterministic program halts if every sequence of choices leads to halting.

So, a nondeterministic program rejects its argument if every sequence of choices results in its halting and returning false.

A Simple Nondeterministic Program

Suppose `range(i, j)` returns the Javascript array `[i, i+1, ..., j]`.

This function tests for non-primality.

```javascript
function nonprime(n)
{
    var p = guess range(2, sqrt(N));
    return (n % p === 0);
}
```

If it can possibly return true, it will do so.
It will always halt, so of all possibilities return false, so will the function.