Welcome! RC 2020. Oslo in Norway. RC 2020. Welcome!

Reversible Programming Languages Capturing Complexity Classes

Lars Kristiansen

Department of Informatics, University of Oslo

Department of Mathematics, University of Oslo

THE SYNTAX OF RBS

$$X \in Variable$$
 ::= $X_1 \mid X_2 \mid X_3 \mid ...$
 $com \in Command$::= $X^+ \mid X^- \mid (X \text{ to } X) \mid com; com$
 $\mid loop X \{ com \}$

The syntax of the language RBS. The variable X in the loop command is not allowed to occur in the loop's body.

THE SYNTAX OF RBS

$$\begin{array}{lll} \textit{X} \in \textbf{Variable} & ::= & \texttt{X}_1 \mid \texttt{X}_2 \mid \texttt{X}_3 \mid \dots \\ \textit{com} \in \textbf{Command} & ::= & \textit{X}^+ \mid \textit{X}^- \mid (\textit{X} \texttt{to} \textit{X}) \mid \textit{com}; \textit{com} \\ & \mid & \texttt{loop} \; \textit{X} \; \{ \; \textit{com} \; \} \\ \end{array}$$

The syntax of the language RBS. The variable X in the loop command is not allowed to occur in the loop's body.

Why RBS \dots ?

THE SYNTAX OF RBS

$$\begin{array}{lll} \textit{X} \in \textbf{Variable} & ::= & \texttt{X}_1 \mid \texttt{X}_2 \mid \texttt{X}_3 \mid \dots \\ \textit{com} \in \textbf{Command} & ::= & \textit{X}^+ \mid \textit{X}^- \mid (\textit{X} \texttt{to} \textit{X}) \mid \textit{com}; \textit{com} \\ & \mid & \texttt{loop} \; \textit{X} \; \{ \; \textit{com} \; \} \\ \end{array}$$

The syntax of the language RBS. The variable X in the loop command is not allowed to occur in the loop's body.

Why RBS ...?

... Reversible Bottomless Stack programs ...

EXAMPLE PROGRAM

Program:	Comments:
	$(* X1 = \langle m, 0^*] *)$
X_1 to X_9 ;	(* the top elements of X_9 is $m *$)
X_2^+ ;	(* $X_1 = \langle 0^*]$ and $X_2 = \langle 1, 0^*]$ *)
$\texttt{loop} \; \texttt{X}_9 \; \big\{$	(* repeat m times *)
X_1 to X_3 ;	
X_2 to X_1 ;	(* swap the top elements of X_1 and X_2 *)
X_3 to X_2 $\}$	

The program accepts every even number and rejects every odd number.

Each program variable \mathtt{X}_i holds a $bottomless\ stack$

$$\langle x_1,\ldots,x_n,0^* \rangle$$
.

Each program variable X_i holds a bottomless stack

$$\langle x_1,\ldots,x_n,0^* \rangle$$
.

 x_1, \ldots, x_n are natural numbers

Each program variable \mathtt{X}_i holds a $bottomless\ stack$

$$\langle x_1,\ldots,x_n,0^* \rangle$$
.

 x_1, \ldots, x_n are natural numbers

 x_1 is the top element of the stack

Each program variable X_i holds a bottomless stack

$$\langle x_1,\ldots,x_n,0^* \rangle$$
.

a stack has no bottom: $\langle x_1, \dots, x_n, 0^*] = \langle x_1, \dots, x_n, 0, 0, 0, \dots]$

 x_1, \ldots, x_n are natural numbers

 x_1 is the top element of the stack

Each program variable X_i holds a bottomless stack

$$\langle x_1,\ldots,x_n,0^* \rangle$$
.

a stack has no bottom: $\langle x_1, \dots, x_n, 0^*] = \langle x_1, \dots, x_n, 0, 0, 0, \dots]$

 x_1, \ldots, x_n are natural numbers

 x_1 is the top element of the stack

 $\langle 0^* |$ is called the zero stack

(X to Y)

moves the top element of the stack held by ${\tt X}$ to the top of stack held by ${\tt Y},$

(X to Y)

moves the top element of the stack held by X to the top of stack

held by Y, that is
$$\{ X = \langle x_1, \dots, x_n, 0^* | \land Y = \langle y_1, \dots, y_m, 0^* | \}$$

 $\{ X = (x_2, \dots, x_n, 0^*) \land Y = (x_1, y_1, \dots, y_m, 0^*) \}$

(Y to X)

moves the top element of the stack held by X to the top of stack

held by Y, that is
$$\left\{ \ X = \left\langle x_1, \dots, x_n, 0^* \right] \ \land \ Y = \left\langle y_1, \dots, y_m, 0^* \right] \ \right\}$$

(X to Y)

 $\{ X = \langle x_2 \dots, x_n, 0^* \} \land Y = \langle x_1, y_1, \dots, y_m, 0^* \} \}$

 $\{ X = \langle x_1, \dots, x_n, 0^* \} \land Y = \langle v_1, \dots, v_m, 0^* \} \}$

$$X^+$$
 (modified successor)

increases the top element of the stack held by ${\tt X}$ by 1 (mod b), that is

$$\{ X = \langle x_1, \dots, x_n, 0^*] \} X^+ \{ X = \langle x_1 + 1 \pmod{b}, x_2 \dots, x_n, 0^*] \}.$$

$${\tt X}^-$$
 (modified predecessor)

decreases the top element of the stack held by X by $1 \pmod{b}$, that is

$$\{ X = \langle x_1, \dots, x_n, 0^*] \} X^- \{ X = \langle x_1 - 1 \pmod{b}, x_2 \dots, x_n, 0^*] \}.$$

Fix a natural number b > 1.

is a reversible operation.

$$\ldots 2, 3, 4, \ldots, b-1, 0, 1, 2, \ldots, b-1, 0, 1, 2, \ldots$$

Fix a natural number b > 1.

To count modulo b

$$\ldots 2, 3, 4, \ldots, b-1, 0, 1, 2, \ldots, b-1, 0, 1, 2, \ldots$$

is a reversible operation.

0 becomes the successor of b-1

b-1 becomes the predecessor of 0

How is this *b* determined?

How is this **b** determined?

The input to a program is a single natural number m.

How is this b determined?

The input to a program is a single natural number m.

When the execution of the program starts, we have by convention

$$X_1 = \langle m, 0^*]$$

where m is the input. All other variables $(X_2, X_3, ...)$ hold the zero stack $(0^*]$.

How is this b determined?

The input to a program is a single natural number m.

When the execution of the program starts, we have by convention

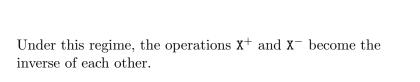
$$X_1 = \langle m, 0^* \rangle$$

where m is the input. All other variables $(X_2, X_3, ...)$ hold the zero stack $\langle 0^* |$.

The base of execution b is set to

$$b := \max(m+1,2)$$

and is kept fixed during the entire execution.



Under this regime, the operations \mathtt{X}^+ and \mathtt{X}^- become the inverse of each other.

We have

$$\{ X = \langle x_1, \dots, x_n, 0^* \} \} X^+; X^- \{ X = \langle x_1, x_2, \dots, x_n, 0^* \} \}.$$

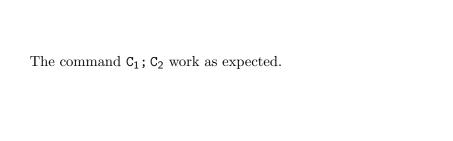
Under this regime, the operations X⁺ and X⁻ become the inverse of each other.

We have

$$\{ X = \langle x_1, \dots, x_n, 0^* \} \} X^+; X^- \{ X = \langle x_1, x_2, \dots, x_n, 0^* \} \}.$$

We have

$$\{ X = \langle x_1, \dots, x_n, 0^* \} \} X^-; X^+ \{ X = \langle x_1, x_2, \dots, x_n, 0^* \} \}.$$



The command C_1 ; C_2 work as expected.

This is the standard composition of the commands C_1 and C_2 , that is, first C_1 is executed, then C_2 is executed.

loop X { C }

executes the command ${\tt C}$ repeatedly k times in a row where k is the top element of the stack held by ${\tt X}$.

loop X { C }

executes the command C repeatedly k times in a row where k is the top element of the stack held by X.

Note that the variable X is not allowed to occur in C and, moreover, the command will not modify the stack held by X.

Definition. We define the reverse command of C, written C^R , inductively over the structure C:

- $(X_i^+)^R = X_i^-$
- $(X_i^-)^R = X_i^+$
- $(X_i \text{ to } X_i)^R = (X_i \text{ to } X_i)$
- $(C_1; C_2)^R = C_2^R; C_1^R$

• $(loop X_i \{C\})^R = loop X_i \{C^R\}.$

Let C be a program, and let X_1, \ldots, X_n be the variables occurring in C. Furthermore, let m be any natural number. We have

$$\{ X_1 = \langle m, 0^*] \land \bigwedge_{i=2}^n X_i = \langle 0^*] \}$$

$$\{X_1 = \langle m, 0^* \rangle \land \bigwedge_{i=2} X_i = \langle 0^* \rangle \}$$

$$C; C^R$$

 $\{ X_1 = \langle m, 0^* | \land \bigwedge_{i=2}^n X_i = \langle 0^* | \}$

Let C be a program, and let X_1, \ldots, X_n be the variables occurring in C. Furthermore, let m be any natural number. We have

$$\left\{ \begin{array}{l} X_1 = \langle m, 0^*] \ \wedge \ \bigwedge_{i=2}^n \ X_i = \langle 0^*] \ \right\}$$

$$C; C^R$$

$$\left\{ \begin{array}{l} X_1 = \langle m, 0^*] \ \wedge \ \bigwedge_{i=2}^n \ X_i = \langle 0^*] \ \right\}$$

The theorem is proved by induction over the structure of C. A detailed proof can be found in my paper.

The considerations above show that we have a programming language that is reversible in a very strong sense.

The considerations above show that we have a programming language that is reversible in a very strong sense.

The next theorem says something about the expressive power of this reversible language.

 $\mathcal{S} = \textit{ETIME}$

What does this theorem say?

 $\mathcal{S} = \textit{etime}$

What is S?

What is ETIME ?

What does this theorem say?

 $\mathcal{S} = \textit{ETIME}$

What does this

theorem say?

What is S?

What is ETIME ?

Let me explain $\mathcal S$ first.

 $\mathcal{S} = \textit{ETIME}$

 ${\mathcal S}$ is the class of problems decidable by an RBS program.

 $\mathcal{S} = \mathit{ETIME}$

What does this theorem say?

An RBS program C accepts the natural number m if C executed with input m terminates with 0 at the top of the stack hold by X_1 , otherwise, C rejects m.

 $\mathcal{S} = \textit{ETIME}$

A problem is simply a set of natural numbers.

 $\mathcal{S} = \mathit{ETIME}$

What does this theorem say?

A problem is simply a set of natural numbers.

An RBS program C decides the problem A if C accepts all m that belong to A and rejects all m that do not belong to A.

 $\mathcal{S} = \textit{ETIME}$

 ${\mathcal S}$ is the class of problems decidable by an RBS program.

 $\mathcal{S} = \mathit{ETIME}$

What does this theorem say?

ETIME is the class of problems decidable by a deterministic Turing machine in time $O(2^{kn})$ for some constant k (recall that n denotes the length of the input).

 $\mathcal{S} = \textit{ETIME}$

The theorem gives a so-called *implicit* characterization of the complexity class ETIME.

 $\mathcal{S} = \textit{etime}$

The theorem gives a so-called *implicit* characterization of the complexity class ETIME.

What does this theorem say?

Please, let me elaborate.

In my paper I share some thoughts on the relationship between implicit computational complexity and reversible computing.

In my paper I share some thoughts on the relationship between *implicit computational complexity* and *reversible computing*.

Complexity classes like ETIME, P, FP, NP, LOGSPACE, PSPACE, and so on, are defined by imposing explicit resource bounds on a particular machine model, namely the Turing machine.

In my paper I share some thoughts on the relationship between *implicit computational complexity* and *reversible computing*.

Complexity classes like ETIME, P, FP, NP, LOGSPACE, PSPACE, and so on, are defined by imposing explicit resource bounds on a particular machine model, namely the Turing machine.

The definitions put constraints on the resources (time, space) available to the Turing machines, but no restrictions on the algorithms available to the Turing machines.

In my paper I share some thoughts on the relationship between *implicit computational complexity* and *reversible computing*.

Complexity classes like ETIME, P, FP, NP, LOGSPACE, PSPACE, and so on, are defined by imposing explicit resource bounds on a particular machine model, namely the Turing machine.

The definitions put constraints on the resources (time, space) available to the Turing machines, but no restrictions on the algorithms available to the Turing machines.

E.g., a Turing machine working in polynomial time may apply any imaginable algorithm (as long as the algorithm can be executed in polynomial time).

Implicit computational complexity theory studies classes of functions (problems, languages) that are defined without imposing explicit resource bounds on machine models, but rather by imposing linguistic constraints on the way algorithms can be formulated.

Implicit computational complexity theory studies classes of functions (problems, languages) that are defined without imposing explicit resource bounds on machine models, but rather by imposing linguistic constraints on the way algorithms can be formulated.

When we explicitly restrict our language for formulating algorithms, that is, our programming language, then we may implicitly restrict the computational resources needed to execute algorithms.

Implicit computational complexity theory studies classes of functions (problems, languages) that are defined without imposing explicit resource bounds on machine models, but rather by imposing linguistic constraints on the way algorithms can be formulated.

When we explicitly restrict our language for formulating algorithms, that is, our programming language, then we may implicitly restrict the computational resources needed to execute algorithms.

If we manage to find a restricted programming language that captures a complexity class, then we will have a so-called implicit characterization.

 $\mathcal{S} = \textit{ETIME}$

The theorem gives an *implicit* characterization of the complexity class ETIME.

There is an obvious link between implicit computational complexity and reversible computing:

A programming language based on natural reversible operations will impose restrictions on the way algorithms can be formulated, and thus, also restrictions on the computational resources needed to execute algorithms.

There is an obvious link between implicit computational complexity and reversible computing:

A programming language based on natural reversible operations will impose restrictions on the way algorithms can be formulated, and thus, also restrictions on the computational resources needed to execute algorithms.

Hence, the following question knocks at the door:

Will it be possible find reversible programming languages that capture some of the standard complexity classes?

There is an obvious link between implicit computational complexity and reversible computing:

A programming language based on natural reversible operations will impose restrictions on the way algorithms can be formulated, and thus, also restrictions on the computational resources needed to execute algorithms.

Hence, the following question knocks at the door:

Will it be possible find reversible programming languages that capture some of the standard complexity classes?

YOU ALREADY KNOW THE ANSWER.

 $\mathcal{S} = \textit{ETIME}$

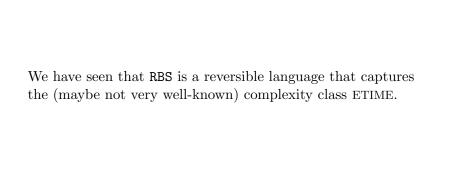
The theorem gives an *implicit* characterization of the complexity class ETIME.

 $\mathcal{S} = \mathit{ETIME}$

What does this theorem say?

The theorem gives an *implicit* characterization of the complexity class ETIME.

The reversible programming language RBS captures the complexity class ETIME.



We have seen that RBS is a reversible language that captures the (maybe not very well-known) complexity class ETIME.

A few small modifications of RBS yield a reversible language RBS, that captures the very well-known complexity class P.

 $\mathcal{S}' = P$

P is the set of problems decidable in polynomial time on a deterministic Turing machine.

S' = P

What does this theorem say?

P is the set of problems decidable in polynomial time on a deterministic Turing machine.

P is considered to be a very good approximation to class of efficiently solvable problems (the problems practically solvable in real life).

S' = P

 \mathcal{S}' is the set of problems decidable by an RBS' program.

$\mathsf{Theorem}$

S' = P

What does this theorem say?

 \mathcal{S}' is the set of problems decidable by an RBS' program. RBS' is a reversible programming

RBS' is a reversible programming language which should be considered as a variant of RBS...slightly more complicated, but still very similar.

EXAMPLE RBS' PROGRAM

```
Program:
                                  Comments:
X_2^-
                                  (* the top element of X_2 is b-1*)
loop X<sub>2</sub> {
                                  (* repeat b-1 times *)
case inp[X_3]=b:
                                  (* X_3 is a pointer into the input *)
  \{X_1 \text{ to } X_9;
                                  (* X_1 holds the zero stack *)
    X_1^+
                                  (* top element of X_1 is 1 *)
   };
X_3^+
                                  (* move pointer to the right *)
};
                                  (* \text{ end of loop } *)
                                  (* top element of X_3 is b-1*)
case inp[X_3]=a:
  \{ X_1 \text{ to } X_9; X_1^+ \}
```

Goodbye! RC 2020. Oslo, Norway. RC 2020 Goodbye!

Thanks for your attention!

??????????????

 \ldots well, may be I have time for a few more \ldots