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

$$
\begin{array}{rcl}
X \in \textbf{Variable} & ::= & \mathtt{X}_1 \mid \mathtt{X}_2 \mid \mathtt{X}_3 \mid \ldots \\
\mathit{com} \in \textbf{Command} & ::= & X^{+} \mid X^{-} \mid (X \,\mathtt{to}\, X) \mid \mathit{com}\,;\,\mathit{com} \\
 & \mid & \mathtt{loop}\, X \,\{\, \mathit{com} \,\}
\end{array}
$$

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

THE SYNTAX OF RBS

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

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 . . . ?

THE SYNTAX OF RBS

$$
\begin{array}{rcl}
X \in \textbf{Variable} & ::= & \mathtt{X}_1 \mid \mathtt{X}_2 \mid \mathtt{X}_3 \mid \ldots \\
\mathit{com} \in \textbf{Command} & ::= & X^+ \mid X^- \mid (X\ \mathtt{to}\ X) \mid \mathit{com}\,;\ \mathit{com} \\
& \mid & \mathtt{loop}\ X\ \{\ \mathit{com}\ \}
\end{array}
$$

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 . . . ?

. . . **R**eversible **B**ottomless **S**tack programs . . .

EXAMPLE PROGRAM

Program:	*Comments:*
	(* $\mathtt{X}_1 = \langle m, 0^*]$ *)
$\mathtt{X}_1$ to $\mathtt{X}_9$;	(* the top elements of $\mathtt{X}_9$ is m *)
$\mathtt{X}_2^+$;	(* $\mathtt{X}_1 = \langle 0^*]$ and $\mathtt{X}_2 = \langle 1, 0^*]$ *)
loop $\mathtt{X}_9$ {	(* repeat m times *)
$\mathtt{X}_1$ to $\mathtt{X}_3$;	
$\mathtt{X}_2$ to $\mathtt{X}_1$;	(* swap the top elements of $\mathtt{X}_1$ and $\mathtt{X}_2$ *)
$\mathtt{X}_3$ to $\mathtt{X}_2$ }	

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

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

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

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

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

$x_1, \ldots, x_n$ are natural numbers

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

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

$x_1, \ldots, x_n$ are natural numbers

x_1 is the top element of the stack

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

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

$x_1, \ldots, x_n$ are natural numbers

x_1 is the top element of the stack

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

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

$$\langle x_1, \ldots, x_n, 0^*] \; .$$

$x_1, \ldots, x_n$ are natural numbers

x_1 is the top element of the stack

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

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

The command

`(X to Y)`

moves the top element of the stack held by `X` to the top of stack held by `Y`,

The command

$$(\mathtt{X}\ \mathtt{to}\ \mathtt{Y})$$

moves the top element of the stack held by `X` to the top of stack held by `Y`, that is

$$\{\ \mathtt{X} = \langle x_1, \ldots, x_n, 0^*]\ \wedge\ \mathtt{Y} = \langle y_1, \ldots, y_m, 0^*]\ \}$$
$$(\mathtt{X}\ \mathtt{to}\ \mathtt{Y})$$
$$\{\ \mathtt{X} = \langle x_2 \ldots, x_n, 0^*]\ \wedge\ \mathtt{Y} = \langle x_1, y_1, \ldots, y_m, 0^*]\ \}$$

The command

$$(\mathtt{X\ to\ Y})$$

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

$$
\begin{array}{l}
\{\ \mathtt{X} = \langle x_1, \ldots, x_n, 0^*]\ \wedge\ \mathtt{Y} = \langle y_1, \ldots, y_m, 0^*]\ \} \\
(\mathtt{X\ to\ Y}) \\
\{\ \mathtt{X} = \langle x_2 \ldots, x_n, 0^*]\ \wedge\ \mathtt{Y} = \langle x_1, y_1, \ldots, y_m, 0^*]\ \} \\
(\mathtt{Y\ to\ X}) \\
\{\ \mathtt{X} = \langle x_1, \ldots, x_n, 0^*]\ \wedge\ \mathtt{Y} = \langle y_1, \ldots, y_m, 0^*]\ \}
\end{array}
$$

The command

$$\mathtt{X}^{+} \qquad \text{(modified successor)}$$

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

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

The command

$$\mathtt{X}^- \qquad \text{(modified predecessor)}$$

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

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

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.

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

A program will be *executed in base* $b > 1$.

How is this b determined?

A program will be *executed in base* $b > 1$.

How is this b determined?

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

A program will be *executed in base* $b > 1$.

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

$$\mathtt{X}_1 \;=\; \langle m, 0^*]$$

where m is the input. All other variables $(\mathtt{X}_2, \mathtt{X}_3, \ldots)$ hold the zero stack $\langle 0^*]$.

A program will be *executed in base* $b > 1$.

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

$$\mathtt{X}_1 = \langle m, 0^*]$$

where m is the input. All other variables ($\mathtt{X}_2, \mathtt{X}_3, \ldots$) 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.

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

We have

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

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

We have

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

We have

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

The command $\texttt{C}_1\texttt{;}\,\texttt{C}_2$ work as expected.

The command $\texttt{C}_1\texttt{;}\,\texttt{C}_2$ work as expected.

This is the standard composition of the commands $\texttt{C}_1$ and $\texttt{C}_2$, that is, first $\texttt{C}_1$ is executed, then $\texttt{C}_2$ is executed.

The command

$$\texttt{loop}\;\texttt{X}\;\{\;\texttt{C}\;\}$$

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

The command

$$\texttt{loop}\,\texttt{X}\,\{\,\texttt{C}\,\}$$

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 $\texttt{C}$, written $\texttt{C}^R$, inductively over the structure $\texttt{C}$:

- $(\texttt{X}_i^+)^R = \texttt{X}_i^-$
- $(\texttt{X}_i^-)^R = \texttt{X}_i^+$
- $(\texttt{X}_i\,\texttt{to}\,\texttt{X}_j)^R = (\texttt{X}_j\,\texttt{to}\,\texttt{X}_i)$
- $(\texttt{C}_1\,;\,\texttt{C}_2)^R = \texttt{C}_2^R\,;\,\texttt{C}_1^R$
- $(\texttt{loop}\,\texttt{X}_i\,\{\,\texttt{C}\,\})^R = \texttt{loop}\,\texttt{X}_i\,\{\,\texttt{C}^R\,\}$.

Theorem

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

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

$$C;\, C^R$$

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

Theorem

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

$$\{\ X_1 = \langle m, 0^*] \ \wedge\ \bigwedge_{i=2}^{n}\ X_i = \langle 0^*]\ \}$$
$$C;\ C^R$$
$$\{\ X_1 = \langle m, 0^*] \ \wedge\ \bigwedge_{i=2}^{n}\ X_i = \langle 0^*]\ \}$$

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.

Theorem

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

What does this theorem say?

Theorem

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

What is $\mathcal{S}$?

What is ETIME ?

What does this theorem say?

Theorem

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

What does this theorem say?

What is $\mathcal{S}$?

What is ETIME ?

Let me explain $\mathcal{S}$ first.

Theorem

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

What does this theorem say?

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

Theorem

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

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 $\mathtt{X}_1$, otherwise, `C` *rejects* m.

Theorem

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

What does this theorem say?

A *problem* is simply a set of natural numbers.

Theorem

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

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.

Theorem

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

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

What does this theorem say?

Theorem

$$\mathcal{S} = ETIME$$

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).

Theorem

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

What does this theorem say?

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

Theorem

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

What does this theorem say?

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

Please, let me elaborate.

Reversible Computing and Implicit Computational Complexity

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

Reversible Computing and Implicit Computational Complexity

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).

Reversible Computing and Implicit Computational Complexity

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.

Reversible Computing and Implicit Computational Complexity

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.

Reversible Computing and Implicit Computational Complexity

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.

Theorem

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

What does this theorem say?

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

Reversible Computing and Implicit Computational Complexity

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?*

Reversible Computing and Implicit Computational Complexity

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.

Theorem

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

What does this theorem say?

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

Theorem

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

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.

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.

Theorem

$$\mathcal{S}' = P$$

What does this theorem say?

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

Theorem

$$\mathcal{S}' = P$$

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).

Theorem

$$\mathcal{S}' = P$$

What does this theorem say?

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

Theorem

$$\mathcal{S}' = P$$

What does this theorem say?

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

`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:*
$\mathtt{X}_2^-$	(* the top element of $\mathtt{X}_2$ is $b-1$ *)
loop $\mathtt{X}_2$ {	(* repeat $b-1$ times *)
case inp[$\mathtt{X}_3$]=b:	(* $\mathtt{X}_3$ is a pointer into the input *)
{ $\mathtt{X}_1$ to $\mathtt{X}_9$;	(* $\mathtt{X}_1$ holds the zero stack *)
$\mathtt{X}_1^+$	(* top element of $\mathtt{X}_1$ is 1 *)
};	
$\mathtt{X}_3^+$	(* move pointer to the right *)
};	(* end of loop *)
case inp[$\mathtt{X}_3$]=a:	(* top element of $\mathtt{X}_3$ is $b-1$ *)
{ $\mathtt{X}_1$ to $\mathtt{X}_9$; $\mathtt{X}_1^+$ }	

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

Thanks for your attention!

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

…well, maybe I have time for a few more …