Equivalence between URM and URM-Back machines

Stevanato Giacomo

28/11/2022

We define a URM-Back machine as a URM-like machine with the usual $Z(n)$, $S(n)$, $T(n,m)$ and $J(n,m,t)$ instructions, but the $J(n,m,t)$ instruction is restricted to only backward jumps.

We want to show that the set $\mathcal{C}^B$ of URM-Back computable functions is equal to the set $\mathcal{C}$ of computable functions.

($\mathcal{C}^B \subseteq \mathcal{C}$) This is trivial because since every URM-Back program is a valid URM program then every URM-Back computable function is also URM computable by the same program.

($\mathcal{C} \subseteq \mathcal{C}^B$) We know that the set $\mathcal{B}$ of URM computable functions is the same as the set of partial recursive functions. We show that $\mathcal{C}$ is contained in $\mathcal{C}^B$ by showing that for every partial recursive function there exists a URM-Back program that computes it. Since the set of partial recursive functions is the same as the URM computable functions, then this proves that $\mathcal{C} \subseteq \mathcal{C}^B$.
The proofs that the set of URM-Back computable functions contains the zero function, the successor function, the projection function and is closed under composition are the same as for URM program, since they don't involve loops.
To prove that $\mathcal{C}^B$ is also closed under primitive recursion and unbounded minimalization we first define the following constructs:

- EQ(a,b,y)
  This URM-Back construct executes $r_y \leftarrow 1$ if $r_a = r_b$, otherwise it executes $r_y \leftarrow 0$. It doesn't modify the memory used by the rest of the enclosing program. Let $m = \rho(P)$ where P is the enclosing program without this construct. It is equivalent to the following instructions:

$\mathbf{T}$(a , m+1)
$\mathbf{T}$(b , m+2)
$\mathbf{Z}$(m+3)
$\mathbf{S}$(m+1)
LOOP:
$\mathbf{T}$(m+3 , y)
$\mathbf{S}$(m+3)
$\mathbf{S}$(m+2)
$\mathbf{J}$(m+1 , m+2 , LOOP)

When we reach the $J(m+1, m+2, LOOP)$ instruction for the first time we get the following state:

$$\begin{aligned} r_{m+1} &= r_a + 1 \\ r_{m+2} &= r_b + 1 \\ r_{m+3} &= 1 \\ r_y &= 0 \end{aligned}$$

We note that $r_{m+1} = r_{m+2} \iff r_a = r_b$. We distinguish two cases:

- $r_a \neq r_b$ thus the jump is skipped and we get $r_y = 0$ as we wanted;
- $r_a = r_b$ thus the jump is taken and we execute another iteration until the same jump instruction. Then the state becomes:

$$\begin{aligned} r_{m+1} &= r_a + 1 \\ r_{m+2} &= r_b + 2 \\ r_{m+3} &= 2 \\ r_y &= 1 \end{aligned}$$

  r_{m+1} and r_{m+2} are now different thus the jump is not taken. $r_y = 1$ is what we wanted since $r_a = r_b$.

- JUMPF(a, b, y, t)
  This URM-Back construct executes a $J(a, b, t)$ but only if $r_y = 0$, otherwise the execution continues without jumping. It doesn't modify the memory used by the rest of the enclosing program. Let $m = \rho(P)$ where P is the enclosing program without this construct. It is equivalent to the following instructions:

EQ(a , b , m+1)
T(y , m+2)
S(m+2)
J(m+1, m+2, **t**)

When we reach the $J(m+1, m+2, t)$ instruction for the first time we get the following state:

$$r_{m+1} = \begin{cases} 0 & \text{if } r_a \neq r_b \\ 1 & \text{otherwise} \end{cases}$$
$$r_{m+2} = r_y + 1$$

We distinguish two cases:

- $r_a \neq r_b$: then $r_{m+1} = 0 \neq r_y + 1$ and the jump is skipped like we wanted;
- $r_a = r_b$: then $r_{m+1} = 1$ and the jump is taken iff $r_y = 0$, like we wanted.

Primitive recursion Given $f : \mathbb{N}^k \to \mathbb{N}$, $g : \mathbb{N}^{k+2} \to \mathbb{N}$ two computable functions, and let F and G their corresponding URM-Back programs that compute them. We consider the function $h : \mathbb{N}^{k+1} \to \mathbb{N}$ defined by primitive recursion by f and g:

$$h(\vec{x}, 0) = f(\vec{x})$$
$$h(\vec{x}, i+1) = g(\vec{x}, i, h(\vec{x}, i))$$

We want to find a URM-Back program that computes h.

Let $g' : \mathbb{N}^{k+3} \to \mathbb{N}$ the following function:

$$g'(\vec{x}, i, p, y) \begin{cases} g(\vec{x}, i, p) & \text{if } y = 0 \\ \downarrow & \text{otherwise} \end{cases}$$

It is not important what the value of $g'(\vec{x}, i, p, y)$ is if $y \neq 0$, what's important is that it is always defined in that case. We can find a URM-Back program G' for g' by taking the program G and switching every $J(a, b, t)$ into the expansion of $JUMPF(a, b, y, t)$. We already proved that if $y = 0$ then $JUMPF(a, b, y, t)$ is the same as $JUMP(a, b, t)$, thus preserving the execution of G. Otherwise, if $y \neq 0$, it skips the jump, guaranteeing termination

because the program can only execute instructions with increasing indexes and their number is bounded. Finally, we can implement primitive recursion by the following URM-Back program:

```
T(1 , m+1)
 ...
T(k , m+k)
T(k+1, m+k+4)
F[m+1, ... , m+k -> m+k+2]
LOOP:
EQ(m+k+1, m+k+4, m+k+3)
T(m+k+2, 1)
G'[m+1, ... , m+k+3 -> m+k+2]
S(m+k+1)
J(m+k+3, m+k+5, LOOP)
```

Like the implementation of primitive recursion for URM programs, the arguments are moved to scratch registers from $m+1$ to $m+k$. $m+k+1$ stores the current recursion depth and $m+k+2$ the result of the previous function invocation. The maximum recursion depth is however stored in $m+k+4$ since $m+k+3$ is reserved for the value of the guard y. At the beginning of every iteration the invariant is that $m+k+2$ stores the $m+k+1$th recursive call. This is true at the beginnin because it stores the return value of f, which is $h(\vec{x}, 0)$. This is also true after each iteration, since ... TODO

Unbounded minimalization Given $f : \mathbb{N}^{k+1} \to \mathbb{N}$ we want to find a URM-Back program that computes the $h : \mathbb{N}^k \to \mathbb{N}$ function $h(\vec{x}) = \mu i.f(\vec{x}, i)$. Let F a URM-Back program that computes f. The URM-Back program that computes h is:

```
T(1 , m+1)
 ...
T(k , m+k)
LOOP:
F[m+1, ... , m+k+1 -> 1]
EQ(1 , m+k+2, m+k+3)
T(m+k+1, 1)
S(m+k+1)
J(m+k+2, m+k+3, LOOP)
```

Like the implementation of primitive recursion for URM programs, the arguments are moved to scratch registers from $m+1$ to $m+k$. At the start of each iteration, $m+k+1$ stores its index starting from 0. F is executed, and

the result is stored in r_1. Then r_1 is compared to $r_{m+k+2} = 0$, and $m + k + 3$ stores the result, which is 0 if $r_1 \neq 0$ and 1 otherwise. $r_1 = rm + k + 1$ is executed to store the result in register 1, in case the current iteration is the last one, otherwise the next execution of F will overwrite it. r_{m+k+1} is incremented at the end of the iteration to prepare for the next iteration. Finally, $rm + k + 2 = 0$ is confronted with r_{m+k+3}, which was $0 \iff r_1 \neq 0$, that is the function f didn't return 0 and thus another iteration has to start. Otherwise r_1 already holds the correct return value and the program can end.