CENTRAL ASIAN JOURNAL OF MATHEMATICAL THEORY AND COMPUTER SCIENCES
https://cajmtcs.centralasianstudies.org

Volume: 04 Issue: 4 | Apr 2023 ISSN: 2660-5309

BUDGE: A PROGRAMMING LANGUAGE AND A THEOREM PROVER

Boro Sitnikovski
Skopje, North Macedonia
buritomath@gmail.com

ABSTRACT

We present a simple programming language based on Godel numbering and prime factorization, enhanced with explicit, scoped loops, allowing for easy program composition. Further, we will present a theorem prover that allows expressing and working with formal systems. The theorem prover is simple as it relies merely on a substitution rule and set equality to derive theorems. Finally, we will represent the programming language in the theorem prover. We will show the syntax and semantics of both, and then provide a few example programs and their evaluation.

ARTICLEINFO

Article history:
Received 17 Feb 2023
Revised form 18 Mar 2023
Accepted 22 Apr 2023

Keywords: *Programming language, theorem prover, computational model, Godel numbering.*

---

1. Budge programming language

Budge-PL (b'dzh) is a simple programming language. The programming language uses Go¨del numbering[1] to represent registers and their values by relying on the Fundamental Theorem of Arithmetic[2]. For example, to represent the values 1, 2, and 3 in memory, we would calculate $2^1 \cdot 3^2 \cdot 5^3$ (the first three primes 2, 3, 5 to the power of the number of the value at the corresponding register), arriving at the state $i = 2250$. We can extract 1, 2, and 3 from 2250 using prime factorization.

Budge-PL uses similar constructs as FRACTRAN[3]. However, Budge-PL provides a more convenient way to construct loops and uses integers rather than fractions to denote instructions. A negative integer will decrease a register's value, while a positive integer will increase a register's value. In addition, it provides an easy way to code loops by using nested parenthesis[1]. Finally, it abstracts prime numbers in the code from the programmer.

1.1 Syntax and semantics

Where data is represented as $i \in N^+$ (product of primes), the syntax of the code in Backus-Naur form[4] is:

<posn> ::= "1" | "2" | ... <negn> ::= "-1" | "-2" | ...

<stmt> ::= <posn> | <negn> | "("<posn>","<stmts>")"

<stmts> ::= <stmt>","<stmts> | <stmt> <code> ::= "("<stmts>")"

[1] One disadvantage of the syntax for loops is that programs can't be as easily represented in memory.

Let $p(n)$ be the n-th prime number. Let sign(n) = 1 if $n > 0$ and −1 otherwise; this will determine whether we need to multiply or divide. With $\forall x,n_x \in \mathrm{Z} \wedge n_x' \in \mathrm{Z}$, let $E(i,s)$ represent the evaluation of a sequence s (<code>) for input i be defined by:

$$E(i,s) = \begin{cases} E(i \cdot p(|n_0|)^{\mathrm{sign}(n_0)}, (n_1,\ldots,n_k)) & s = (n_0,n_1,\ldots,n_k) \wedge i \cdot p(|n_0|)^{\mathrm{sign}(n_0)} \in \mathbb{N}, \\ E(i,(n_1,\ldots,n_k)) & s = (n_0,n_1,\ldots,n_k) \wedge i \cdot p(|n_0|)^{\mathrm{sign}(n_0)} \notin \mathbb{N}, \\ E(E(i,(n_0',\ldots,n_k')),s) & s = ((P,n_0',\ldots,n_k'),n_0,\ldots,n_j) \wedge i \cdot p(|P|)^{-1} \in \mathbb{N}, \\ E(i,(n_0,\ldots,n_j)) & s = ((P,n_0',\ldots,n_k'),n_0,\ldots,n_j) \wedge i \cdot p(|P|)^{-1} \notin \mathbb{N}, \\ i & \text{otherwise, that is, } s = () \end{cases}$$

Semantically, the first case handles increasing/decreasing a value in a register n0. The second case is for skipping an instruction. The third and fourth cases represent the start and end of a loop (nested parenthesis).

1.2 Example programs

1.2.1 Addition of numbers (evaluation explanation)

To compute $E(2^3 \cdot 3^3,((2,-2,1)))$ we iterate the sequence (−2,1) until $_{i/p(2)}$ is no longer an integer, that is,$\frac{i}{3}$:

1. Initially, $i = 2^3 \cdot 3^3 = 216$, and since $\frac{216}{3} = 72 \in \mathbb{N}$, proceed with evaluation.
2. Calculate $p(|n|)^{\mathrm{sign}(n)}$ for $n = -2$: $i' = p(2)^{-1} = 3^{-1}$. Since $i \cdot i' \in \mathrm{N}$, set i to $i \cdot i' = 216 \cdot i' = 72$.
3. Calculate $p(|n|)^{\mathrm{sign}(n)}$ for $n = 1$: $i' = p(1)^1 = 2$. Since $i \cdot i' \in \mathrm{N}$, set i to $i \cdot i' = 72 \cdot i' = 144$.
4. At this point, we go back and check the condition if $\frac{144}{3} \in \mathbb{N}$- proceed with the evaluation.
5. Calculate $p(|n|)^{\mathrm{sign}(n)}$ for $n = -2$: $i' = p(2)^{-1} = 3^{-1}$. Since $i \cdot i' \in \mathrm{N}$, set i to $i \cdot i' = 144 \cdot i' = 48$.
6. Calculate $p(|n|)^{\mathrm{sign}(n)}$ for $n = 1$: $i' = p(1)^1 = 2$. Since $i \cdot i' \in \mathrm{N}$, set i to $i \cdot i' = 48 \cdot i' = 96$.
7. At this point, we go back and check the condition if $\frac{96}{3} \in \mathbb{N}$- proceed with the evaluation.
8. Calculate $p(|n|)^{\mathrm{sign}(n)}$ for $n = -2$: $i' = p(2)^{-1} = 3^{-1}$. Since $i \cdot i' \in \mathrm{N}$, set i to $i \cdot i' = 96 \cdot i' = 32$.
9. Calculate $p(|n|)^{\mathrm{sign}(n)}$ for $n = 1$: $i' = p(1)^1 = 2$. Since $i \cdot i' \in \mathrm{N}$, set i to $i \cdot i' = 32 \cdot i' = 64$.
10. Now we have that $\frac{64}{3} \notin \mathbb{N}$, so the evaluation halts.

Thus, i is now equal to $64 = 2^6$. That is, the value from the first register $p(1)$ and the value from the second register $p(2)$ were added and then stored in the first register, $p(1)$. In general,$E(2^a \cdot 3^b,((2,-2,1))) = 2^n$, with $n = a + b$.

1.2.2 Other arithmetic operations

Subtraction: $E(2^x \cdot 3^y,s_s) = 2^n \cdot 3^k$ where $n = |x - y|$ and $k = 1$ if $y > x$, and $k = 0$ otherwise.

s_s = ((1,−1,3,5),(2,−2,4,6),(3,−3,−4),(6,−5,−6),(4,−4,1,3),(3,(3,−3),2),(5,−5,1))

Multiplication: $E(2^x \cdot 3^y,s_m) = 2^n$ where $n = x \cdot y$.

s_m = ((1,−1,(2,−2,3,4),(4,−4,2)),(2,−2),(3,−3,1))

Division: $E(2^a \cdot 3^d,s_d) = 2^q \cdot 3^r$ where $a = qd + r$ and $0 \leq r < d$.

s_d = ((2,−2,7),(1,(7,−7,2,8),(8,−8,7))++ s_s++

(9,(2,−2,(1,−1,−7),(7,−7,8),−9)),(7,−7),(9,−9,1),(8,−8,2))

1.3 Composing and interpreting programs

As we saw with s_d, sequences can be composed by concatenating them: $\forall s_1,\forall s_2,E(E(i,s_1),s_2) = E(i,s_1 ++ s_2)$. For example, the sequence (1,2,2,(2,−2,1)) is consisted of concatenating (1,2,2) and ((2,−2,1)); increasing the first and the second register by 1 and 2 respectively, and then add the registers together, storing the result in the first register.

We show the pseudo-code representation of this sequence by following its semantical interpretation:

r1 += 1; r2 += 2; // sequence 1

while (r2 > 0) { r2 -= 1; r1 += 1; } // sequence 2

// r1 += r2; r2 = 0; // sequence 2 optimized

2. Budge theorem prover

Budge-TP (b'dzh) is a theorem prover that allows expressing formal systems. Formal systems are important because they lie at the core of mathematics. It is directly inspired by Prolog[5], where the main difference is that there is no automated deduction and every step has to be manually specified. This allows for a more explicit understanding of formal systems.

Budge-TP has a small Trusted-Computing Base (TCB). Its semantics rely only on substitution and equality check (symbol comparison), though they are still powerful enough to represent any formal system, including computation, as we will see next.

2.1 Semantics

Within Budge-TP, a formal system is defined by the tuple $F = (R,V,T)$ together with the functions $subst^{n}_{rule}$ and $subst^{n}_{thm}$ where $R_n \in R$ is a set of rules of n-ary arguments, V is a set of variables, and T is a set of theorems. A rule $r = (r_1,...,r_n) \in R_n$ is a sequence of string of symbols; it can be roughly interpreted as a function $r_1 \rightarrow ... \rightarrow r_n$, where the n-th argument represents a conclusion, and the others represent hypotheses.

Let $S \subseteq V \times T$ denote a set of substitutions, and $X[t/v]$ denote the expression X in which each occurrence of v is replaced with t. We define the following function which performs substitution on a rule's hypotheses and conclusion:

$$subst^{n}_{rule}(r,S) = \begin{cases} subst^{n}_{rule}(r_1[t/v],\ldots,r_n[t/v], S \setminus \{(v,t)\}), & r = (r_1,\ldots,r_n) \wedge (v,t) \in S \\ r & S = \emptyset \end{cases}$$

Let $h = (h_1,...,h_n{-}_1)$ where $\forall i,h_i \in T$. The function $subst_{thm}{}^{n-1}(h,S)$ is defined similarly. For deriving new theorems, we say that $t = subst^{1}_{rule}((r_n),S) \in T$ (i.e., t is a theorem) if and only if:

$subst_{rule}{}^{n-1}((r_1,...,r_n{-}1),S) = subst_{thm}{}^{n-1}(h,S)$

Terms and axioms are represented as 1-ary rules; note that for $n = 1$ we have $subst^{0}_{rule}((),S) = () = subst^{0}_{thm}((),S)$ i.e. all 1-ary rules are theorems: $\forall r,r \in R_1 \rightarrow r \in T$.

2.2 Syntax

Even though we used set theory[6] to represent the semantics, we can liberate from set theory and use a more convenient syntax. Every statement is of the form:

r<name> : <expr> [-> <expr> [-> ... -> <expr>]] t<name> : <ruleN> [x=X;y=Y;...] [arg1] [arg2] [...] [argn]

The syntax r<name> specifies a rule, and t<name> specifies a theorem. For <name> and <expr>, any string of characters is accepted except ':' and ' ' (whitespace) for <name> and '->' for <expr>. Square brackets represent optional values. Lowercase characters in a rule expression are considered a variable and will be used for substitution within the expressions.

In a rule, all expressions but the last are considered the hypothesis (arguments to be passed when used in a theorem), and the last is the conclusion. For theorems, the rule <ruleN> will be applied to the corresponding arguments. Substitution with theorems (x with theorem X; y with theorem Y...) will be performed in both the rule's hypotheses and the theorem's provided argument, and they will be matched/unified. If unification is successful, the final argument in the rule argn will be the result.

2.3 Example theorems

2.3.1 MIU system[1] (set theoretical syntax)

Let R = {{⊢ MI,I},{(⊢ Mx,⊢ Mxx)}}, V = {x}. The particular choice of R_1 allows us to pick S = {(x,I)}; since I is a 1-ary rule, I ∈ T. Similarly, ⊢ MI ∈ T. To prove ⊢ MII ∈ T, we use the rule within R_2 and since (x,I) ∈ S, we get that $subst^1_{rule}$((⊢ Mx),S) =⊢ MI = $subst^1_{thm}$((⊢ MI),S). Since the rule's arguments match the theorem's hypotheses, $subst^1_{rule}$((⊢ Mxx),S) =⊢ MII ∈ T.

2.3.2 MIU system (Budge-TP syntax)

With the following code, we define the terms, the initial axiom and the rules of inference, and a few example theorems:

```
# Terms
rTmM : M
rTmI : I
rTmU : U
tmM! : rTmM
tmI! : rTmI
tmU! : rTmU
rTmxy : xy
```

```
# Axiom and rules
rMI : |- MI
thMI : rMI
r1 : |- xI -> |- xIU
r2 : |- Mx -> |- Mxx
r3 : |- xIIIy -> |- xUy
```

```
# Example theorems thMII : r2 x=tmI! thMI

tmII! : rTmxy x=tmI!;y=tmI!
thMIIII : r2 x=tmII! thMII
thMUI : r3 x=tmM!;y=tmI! thMIIII
```

The theorems, once deduced, produce the following results:

thMI : |- MI

thMII : |- MII

thMIIII : |- MIIII

thMUI : |- MUI

2.3.3 Budge-PL language (Budge-TP syntax)

Budge-PL uses number theory and prime numbers to store data as registers. We represent a two-register Budge-PL within Budge-TP as a lower system that does not rely on number theory, but rather on a few basic rules:

```
# Lists and numbers
rMkList : (x y)
rTmNil : NIL
rTm0 : 0
rTmS : Sx rTmP : Px

# Initial program
rInitState : p (a b)
```

```
# Commands 1, -1, 2, -2 respectively
rNextState+1 : (S0 x) (a b) -> x (Sa b)
rNextState-1 : (P0 x) (Sa b) -> x (a b)
rNextState+2 : (SS0 x) (a b) -> x (a Sb)
rNextState-2 : (PP0 x) (a Sb) -> x (a b)

# Commands for looping on the second register
rLoop2Base : ((SS0 x) y) (a 0) -> y (a 0)
rLoop2Succ : ((SS0 x) y) (a Sb)
-> APPEND x ((SS0 x) y) z -> z (a Sb)
```

Together with the following helper rules for appending lists:

```
# Appending lists
rAppendNil : APPEND NIL y y
rAppendRec : APPEND x y z -> APPEND (a x) y (a z)
```

To calculate the program ((2,−2,1)) with the first register 1 and the second 2, we use the rules in order: rInitState, rLoop2Succ, rNextState-2, rNextState+1, rLoop2Succ, rNextState-2, rNextState+1, rLoop2Base. These rules take the initial register state (1 2) and evaluate it to (3 0) which represents the sum.

References

1. Hofstadter, Douglas R. *Godel, Escher, Bach: An Eternal Golden Braid¨*. New York: Basic books, 1979.
2. Korniłowicz, Artur, and Piotr Rudnicki. ”Fundamental Theorem of Arithmetic.” *Formalized Mathematics* 12, no. 2 (2004): 179-186.
3. Conway, John H.”Fractran: A simple universal programming language for arithmetic.” In *Open problems in Communication and Computation*, pp. 4-26. Springer, New York, NY, 1987.
4. McCracken, Daniel D., and Edwin D. Reilly. ”Backus-Naur Form (BNF).” In *Encyclopedia of Computer Science*, pp. 129-131. 2003.
5. Clocksin, William F., and Christopher S. Mellish. *Programming in Prolog*. Springer Science & Business Media, 2003.
6. Velleman, Daniel J. *How to Prove It: A Structured Approach*. Cambridge University Press, 2019.
7. Sitnikovski, B. Budge programming language and theorem prover (Python implementation). [Online]. Available: https://github.com/bor0/budge/ (Accessed Aug. 2022)