On the expressive power of the Loop language

T. Crolard, S. Lacas and P. Valarcher

LACL - University of Paris 12 - France : crolard@univ-paris12.fr
Trusted Logic - France : Samuel.Lacas@trusted-logic.fr
LIFAR - University of Rouen - France : Pierre.Valarcher@univ-rouen.fr

October, 2005

1. Introduction. This paper is devoted to the study of the expressive power of an elementary imperative programming language similar to the Loop language described by Meyer and Ritchie in [MR76].

While the λ-calculus is usually used to describe the denotational semantics of programming languages, we exploit it to encode the *operational semantics* of this imperative language. More precisely, we define a lock step simulation of imperative programs by pure functional programs (λ-terms with natural numbers and recursor). This simulation enables us to derive properties concerning this imperative language from previously known results in the λ-calculus community (see for instance [Col91] for such results).

The main property which is derived using this framework is the following : there is no program in the Loop language which computes the minimum of two natural numbers n and m in time *O(min(n,m))*. In order to prove this result, we first need to generalize a result by Colson and Fredholm called the "ultimate obtinacy" property in [CF98].

The plan of the paper is the following. Section 3 is devoted to the presentation of our target functional language which corresponds to Gödel system T equipped with a primitive predecesor function, products and the call-by-value strategy. In section 4, we sketch the proof of the "ultimate obtinacy" for our extended system. Our version of the Loop-language is described in section 5 and the lock-step simulation is eventually presented in details in section 6.

2. Related works. In his pioneering work [Col91], Colson applies denotational semantics (with the domain of lazy integers [Esc93]) and proves that although the function that computes the minimum of two natural numbers is obviously primitive recursive (see for instance [Pet68] for a formal definition), there is no way to implement it efficiently as a *primitive recursive algorithms* (represented by PR-combinators in [Col91]). His main theorem states the *ultimate obstinacy* property of PR-combinators. Informally, this property express the fact that once an algorithm begin to consume an argument, it will never switch to another argument (a constructive proof of this property by Coquand may be found in [Coq92]). This idea has been developed further by David [Dav01] who defines a trace semantics which allows him to prove a stronger property (the *backtracking property*). The third author has proved in [Val96] similar results about intensionnal behiavour of other primitive recursive schemes. At the same time, L. Colson and D. Fredholm [Fre96, CF98] have extended these results to Gödel's system T equipped with a call-by-value strategy (and recursion over lists of natural numbers). Related issues concerning non-determinism and sequentiality have been studied by Brookes and Dancannet [BD95, DB96].

More recently, Moschovakis [Mos03] established linear lower bounds for the complexity of non-trivial primitive recursive algorithm from *piecewise linear* given functions. For instance, he proves that Stein's algorithm for the greatest common divisor cannot be implemented efficiently by a primitive recursive algorithm. Eventually, a partial solution to an open problem concerning the classical *Euclidan algorithm* mentionned in [Mos03] has been found by Van Den Dries [Dri03].

3. Gödel's system T under call-by-value. In this section, we extend the Gödel's system T with product types (tuples and projections) and a constant-time predecessor operation. The resulting system is thus closer to some realistic programming language. Its formal definition is summed up in table 1. The reduction rules presented here implements the so-called weak reduction call-by-value strategy (since we do not reduce under an abstaction). Although we do not recall the type system (for lack of space), we shall only consider well-typed terms in the sequel. As usual, we shall consider the form $\mathbf{let}\, x = u\, \mathbf{in}\, t$ as an abbreviation for $(\lambda x.t\ u)$.

Syntax			*Evaluation contexts*		
t, u	::=	0	$C[\,]$	::=	$[\,]$
	|	$S(t)$		|	$(C[\,]\ t)$
	|	$\mathbf{pred}(t)$		|	$(v\ C[\,])$
	|	$\lambda x.t$		|	$\langle C[\,], t\rangle$
	|	$(t\ u)$		|	$\langle v, C[\,]\rangle$
	|	$\langle t, u\rangle$		|	$S(C[\,])$
	|	$\pi_i(t)$		|	$\mathbf{pred}(C[\,])$
	|	$\mathbf{rec}(t,t,t)$		|	$\mathbf{rec}(C[\,], t_2, t_3)$
Values				|	$\mathbf{rec}(v_1, C[\,], t_3)$
v	::=	0		|	$\mathbf{rec}(v_1, v_2, C[\,])$
	|	$S^k(0)$	*Evaluation rules*		
	|	$\lambda x.t$	$C[\lambda x.t\ v]$	$\Rightarrow$	$C[t[v/x]]$
Types			$C[\pi_i\langle v_1, v_2\rangle]$	$\Rightarrow$	$C[v_i]$
T	::=	N	$C[\mathbf{pred}(0)]$	$\Rightarrow$	$C[0]$
	|	$T \to T$	$\boldsymbol{C}[\mathbf{pred}(S(v))]$	$\Rightarrow$	$C[v]$
	|	$T \times T$	$C[\mathbf{rec}(0, v_2, v_3)]$	$\Rightarrow$	$C[v_2]$
			$C[\mathbf{rec}(S(v_1), v_2, v_3)]$	$\Rightarrow$	$C[(v_3\ v_1\ \mathbf{rec}(v_1, v_2, v_3))]$

Table 1. Types, terms, values and evaluation rules

4. Ultimate obstinacy. We show here that the ultimate obstinacy property holds for our (extended) call-by-value system T. We sketch a (new) direct proof of this result (we rely here on two important properties hold for Gödel's system T, namely strong normalization of well-typed terms and the Church-Rosser property).

Definition 1. *We say that a term t is in $\perp$-form iff one of the following conditions (1-5) holds: (1) t is a variable, (2) $t = t'\,u$ with t' in $\perp$-form, (3) $t = \pi_i(t')$ with t' in $\perp$-form, (4) $t = \mathbf{rec}(t', u, v)$ with t' in $\perp$-form, (5) $t = \mathbf{pred}(t')$ with t' in $\perp$-form.*

Lemma 2. *Let t be a weak normal term of type N, $U_1 \times U_2$ or $T \to U$ then t is respectively of the form $S^k(t')$ with $t' = 0$ or t' in $\perp$-form, $\langle t_1, t_2\rangle$ with t_1, t_2 in $\perp$-form and $\lambda x.t'$ with t' in $\perp$-form.*

Definition 3. *We say that a term of type N is trivial if its weak normal form is either of the form $S^k(0)$ or of the form $S^k(\mathbf{pred}^n(x_i))$.*

Remark 4. These terms represent constant functions or functions of the form $x_i \mapsto (x_i \dot{-} n) + k$.

Lemma 5. *Given term t with free variables $x_1, ..., x_n$, if $t \rightsquigarrow u$ then $t[S(x_i)/x_i] \rightsquigarrow u[S(x_i)/x_i]$.*

Lemma 6. (unique decomposition) *If t is a term then either t is a value or there exists a unique evaluation context $C[\,]$ and a redex r such that $t = C[r]$.*

Lemma 7. *For any evaluation context $C[\,]$, there exists $C'[\,]$ such that $C[\mathbf{rec}(S^k(0), v_2, v_3)] \rightsquigarrow C'[\mathbf{rec}(S^{k-1}(0), v_2, v_3)]$ and $C'[\,]$ is again an evaluation context.*

Proposition 8. (ultimate obstinacy) *Given a non trivial term t of type N with free variables $x_1, ..., x_n$, if t is in weak normal form then there is some $1 \leqslant i \leqslant n$ such that $t[c_1/x_1, ..., S^k(0)/x_i, ..., c_n/x_n]$ reaches its weak normal form in at least k reductions steps for any terms c_j.*

Corollary 9. *There is no term in Gödel's system T with tuples and constant-time predecessor which computes the minimum of two natural numbers n and m in time $O(min(n, m))$.*

5. The Loop language. The imperative language we consider is a syntactic variant of the Loop language described in [MR76] and [DW83]. Variables (called *registers* in [MR76]) can contain only non-negative integers. The atomic statements are assignments of the form $x_i := c$ (where c is a constant), $x_i := x_j$, $x_i := x_i + 1$ or $x_i := x_i - 1$. The only program constructs are the sequence (a list of statements separated by semi-colons) and the for-loop which implements bounded iteration. A loop has the form **for** $x_i := 1$ **to** $Exp\{Seq\}$ where *Exp* is either a constant or a variable and *Seq* is a sequence.

The operational semantics may be given as a simple abstract machine described as a set of rewriting rules. A rule has the form $state \rightsquigarrow state$, where a state is pair $\langle Program, Environment\rangle$. As usual, an *environment* is a mapping of variables onto values (natural numbers). Note that a code pointer is not required when one uses a semantics of loop unrolling. For instance, here are the rules which concern the loop cosntruct:

$$\langle \textbf{for } x_i := 1 \textbf{ to } Exp\ \{Seq\}, E\rangle \rightsquigarrow \langle \{\}, E\rangle \tag{1}$$

where *Exp* is either the constant 0 or a variable x_j and then $E(x_j) = 0$.

$$\langle \textbf{for } x_i := 1 \textbf{ to } Exp\ \{Seq\}, E\rangle \rightsquigarrow \langle \{\textbf{for } x_i := 1 \textbf{ to } c\ \{Seq\}; x_i := c+1; Seq\}, E\rangle \tag{2}$$

where *Exp* is either the constant $c+1$ or a variable x_j and then $E(x_j) = c+1$.

We are now able to define the semantics of a program (as usual $\rightsquigarrow^\star$ stands for the reflexive and transitive closure of $\rightsquigarrow$).

Definition 10. *Given an initial environment E, we say that a program P evaluates to E' if and only if* $\langle P, E\rangle \rightsquigarrow^\star \langle \{\}, E'\rangle$

6. Lock-step simulation.

Definition 11. *The translation * of a* Loop *program with variables $\vec{x} = x_1, ..., x_k$ into a term is defined by induction on expressions and instructions as follows:*

- $c^* = S^c(0)$
- $x_i^* = x_i$
- $(x_i + 1)^* = S(x_i)$
- $(x_i - 1)^* = \textbf{pred}(x_i)$
- $\{\}^* = \vec{x}$
- $\{I; Seq\}^* = \textbf{let } \vec{x} = I^* \textbf{ in } \{Seq\}^*$ *if I is not an assignment*
- $\{x_i := Exp; Seq\}^* = \textbf{let } x_i = Exp^* \textbf{ in } \{Seq\}^*$
- $(\textbf{for } x_i := 1 \textbf{ to } Exp\ \{\text{Seq}\})^* = \textbf{rec}(Exp^*, \vec{x}, \lambda\vec{x}\lambda x_i\{\text{Seq}\}^*)$
  where Exp is either a constant or a variable

We are now able to state the main theorem (the lock-step simulation) and its corollary concerning the expressive power of the Loop language.

Theorem 12. *If* $\langle I, E\rangle \rightsquigarrow \langle I', E'\rangle$ *then* $I^*[E^*/\vec{x}] \Rightarrow I'^*[E'^*/\vec{x}]$

Proof. By induction on the derivation of $\langle I, E\rangle \rightsquigarrow \langle I', E'\rangle$. □

Corollary 13. *There is no program in the* Loop*-language which computes the minimum of two natural numbers n and m in time O(min(n,m)).*

Bibliography

[BD95] Stephen Brookes and Denis Dancanet. Sequential algorithms, deterministic parallelism, and intensional expressiveness. In proceedings of the *22nd ACM SIGPLAN-SIGACT Symposium on POPL*. pp 13-24. 1995.

[CF98] L. Colson and D. Fredholm. System T, call-by-value and the minimum problem. *Theoretical Computer Science*, 206, pp 301-315. 1998.

[Col91] L. Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 83, pp 57-69, 1991.

[Coq92] T. Coquand. Une preuve directe du théorème d'ultime obstination. *Compte Rendus de l'Académie des Sc.*, 314, Serie I, 1992.

[Dav01] R. David. On the asymptotic behaviour of primitive recursive algorithms. *Theor. Comput. Sci.*, 266(1-2):159–193, 2001.

[DB96] D. Dancanet and S. Brookes. Programming language expressiveness and circuit complexity. In *Internat. Conf. on the Mathematical Foundations of Programming Semantics*, 1996.

[Dri03] L. Van Den Dries. Generating the greatest common divisor, and limitations of primitive recursive algorithms. *to appear in Foundations of Computational Mathematics*, 2003.

[DW83] M. Davis and E. Weyuker. *Computability, Complexity and Languages*. Academic Press, 1983.

[Esc93] Martin Hotzel Escardo. On lazy natural numbers with applications. *SIGACT News*, 24(1), 1993.

[Fre96] D. Fredholm. Computing minimum with primitive recursion over lists. *Theoretical Computer Science*, 163, 1996.

[Mos03] Yiannis N. Moschovakis. On primitive recursive algorithms and the greatest common divisor function. *Theor. Comput. Sci.*, 301(1-3):1–30, 2003.

[MR76] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proc. ACM Nat. Meeting*, 1976.

[Pet68] Roza Peter. *Recursive Functions*. Academic Press, 1968.

[Val96] P. Valarcher. Intensionality vs extensionality and primitive recursion. *ASIAN Computing Science Conference - LNCS*, 1179, 1996.