Computer Programming as Mathematics (Brief description of CL)

Ján Komara and Paul J. Voda

Institute of Informatics, Comenius University Bratislava,
Mlynská dolina, 842 15 Bratislava, Slovakia,
email: {komara,voda}@fmph.uniba.sk

1 Introduction

CL (Clausal Language) is an extremely simple, yet powerful, programming and specification language which is coupled with its own proof system. CL has a precise characterization of both its definable functions and provable properties. The former are exactly the unary primitive recursive functions and the latter correspond exactly to the Π_2-theorems of $I\Sigma_1$-arithmetic which is a rather simple fragment of the well-known Peano arithmetic (see [9]). The fact that the functions of CL are over natural numbers and that their properties are proved in Peano arithmetic is extremely important for its intended application in teaching at the introductory levels of undegraduate studies. We use CL in three courses: *Declarative programming*, *Program verification*, and *Specification of programs and data*. Our experience (and the marks we get from students in teacher evaluations) show that the students readily understand CL. This is primarily because they have a good intuition about the domain of natural numbers.

The simplicity of CL is quite remarkable when compared with the syntax and semantics of the well-known proof systems such as PVS [11], LCF [7], Nuprl [4], Coq [3], HOL [6], OTTER [10], or even of the celebrated Nqthm [1]. This is also true for the data specification part of CL whose utmost simplicity should be contrasted with the relatively verbose syntax and non-trivial semantics of the well-known specification languages such as Larch [8], CLEAR [2], OBJ2 [5], or ASL [12].

2 The Domain

Programs of CL are functions and predicates over the domain of natural numbers. Properties of the programs are expressed in the language of first-order logic interpreted in natural numbers. Proofs are done in a rather weak theory which is a small fragment of Peano arithmetic.

There is hardly a simpler and a better understood domain than natural numbers. Every high school graduate has a very concrete intuition about natural numbers and he can readily visualize functions over them. First-order classical logic with the language containing only one sort of variables and one level of quantification is again the most basic and easiest to understand. Properties of natural numbers, as expressed by axioms of Peano arithmetic, are so obvious that the concept of truth of formulas (interpreted into the standard model of natural numbers) is clear.

This should be contrasted with other proof and specification systems which are typically based on many-sorted algebras (or structures) and which often permit higher order functions. The models are hard to imagine. Interpretation of computable functionals is non-trivial. As a consequence, such systems are almost impossible to teach at the introductory levels as their languages must reflect the complexity of the intended models. Proof systems must be often for higher-order logic. It takes much longer to teach just the syntax. The semantic intuition which, so obvious to the students of CL, is practically non-existent.

On the other hand, a computer scientist when presented with our claim that the domain of natural numbers suffices, sees immediately a possible line of objections. Where are the other types of numbers (integers, rationals)? What about complex data structures for symbolic (non-numeric) computation? Is not the restriction to the natural numbers unrealistic one? Our standard defense is that we are interested only in computable functions because we are doing computer science and thus recursive functions come quite naturally. All other domains (data types) can be coded into natural numbers because otherwise we would not be able to compute functions over them. Our computer scientist then recalls the coding of finite sequences into the exponents of primes and objects to the inefficiency of the coding and to its counter-intuitiveness.

Our answer to the last criticism is that we can identify the domain of natural numbers with the domain of S-expressions of LISP (with the only atom *nil*) by means of a suitable pairing function (*cons*). Every natural number can be then viewed either as generated from 0 by the successor function or from 0 by the pairing function, or any combination of thereof. We answer the question about efficiency of the coding by pointing out that pairing (*cons*) is easily implemented by pointers and that the CL system keeps the numbers either in numeric (binary) notation or as an S-expression and can mix the two of them. Conversions between representations are always possible (adding the number 15 to an S-expression or asking for the head (*car*) of the number 2346789), but they are not exactly cheap because the pairing function is not trivial. However, most programs are well-behaved (one can say that they are implicitly well-typed) and there is rarely a need for such conversions.

We answer the objection about intuitiveness of coding by simply pointing out that easier coding than in LISP is harder to imagine. We have only one atom 0 (*nil*), every positive number is a pair (*cons*). As a consequence, every number can be uniquely viewed as a list of natural numbers and so the coding of finite sequences is extremely simple. Every number is a binary tree (without labels) and trees with labels (union types of the functional programming languages) are easily and very naturally coded (see Sect. 3).

What about the data types such as integers and rationals? Their properties can be encapsulated in a theory and used in a standard intepretation (implementation) where the values are stored in a computer as integers or floating point numbers. This kind of encapsulation is, of course, possible with every standard 'abstract data type' and the user of CL quickly obtains the feeling of working in a many-sorted system while he is writing just unary functions (remember we have pairing) over natural numbers.

3 The Programming Language

The programming language of CL allows the definition of unary functions and predicates (and also of constants) over natural numbers. As an example we show a definition of two well-known functions:

$$\begin{aligned}&\mathit{Fib}(0) = 1\\&\mathit{Fib}(1) = 1\\&\mathit{Fib}(n+2) = \mathit{Fib}(n+1) + \mathit{Fib}(n)\end{aligned}$$

$$\begin{aligned}&\mathit{Rev}(0) = 0\\&\mathit{Rev}(v,w) = \mathit{Rev}(w) \oplus (v,0)\ .\end{aligned}$$

The list reversal function *Rev* takes a number which is either the empty list 0 (*nil*) or a pair v, w and yields the reversed list. Note that we use the infix comma for pairing (*cons*) and abbreviate $s,(t,u)$ to s,t,u. Every natural number is either the atom 0 or a pair v, w for some *smaller* numbers v and w. This is similar to the fact that every number is either 0 or a successor $n+1$ for a smaller number n. We write $s \oplus t$ as an abbreviation for the application $\mathit{App}(x,y)$ of the list concatenation function *App* whis is defined as

$$\begin{aligned}&0 \oplus y = y\\&(v,w) \oplus y = v,(w \oplus y)\quad .\end{aligned}$$

The reader will note that the function *App* is a unary function which takes in the normal case a pair of lists (i.e. numbers) and in the pathological case is defined as $\mathit{App}(0) = 0$ by *default*.

Here is the clausal definition of the predicate *Bt* of being a (code of) binary tree with natural numbers as labels:

$$\begin{aligned}&\mathit{Bt}(E)\\&\mathit{Bt}\ \mathit{Nd}(n,a,b) \leftarrow \mathit{Bt}(a) \wedge \mathit{Bt}(b)\ .\end{aligned}$$

The predicate Bt can be viewed as a CL counter-part of the union type

$type\, Bt = E \mid Nd(N, Bt, Bt)$

well-known from functional programming languages. The *constructors* E (leaf) and Nd (labelled node) are defined in CL as a constant and a function respectively:

$E = 0, 0$
$Nd(x) = 1, x$.

By default, Bt is defined as the minimal predicate satisfying its clauses, i.e. $Bt(n)$ does not hold unless its truth can be established from the clauses. The function Ins, when called as $Ins(n, t)$, yields a binary search tree with the number n inserted into the binary search tree t and it has a CL definition:

$Ins(n, E) = Nd(n, E, E)$
$Ins(n, Nd(m, a, b)) = Nd(m, Ins(n, a), b) \leftarrow n < m$
$Ins(n, Nd(m, a, b)) = Nd(m, a, b) \leftarrow n = m$
$Ins(n, Nd(m, a, b)) = Nd(m, a, Ins(n, b)) \leftarrow n > m$.

There is no typing in CL and $Ins(n, t)$ yields 0 by default if t has not the form of the two variants of binary trees.

Recursive definitions in CL can be used in a checked or unchecked mode. In the unchecked mode CL permits the definitions and execution of exactly all unary partial recursive functions. However, we cannot state and prove any properties of such functions. In the checked mode a recursive definition is accepted only if the user supplies a measure function in which the recursive arguments go down. For instance, the measure function for the function Ins is the tail function T (*cdr*) because we have for the first recursive application:

$$T(n, a) = a < 1, m, a, b = Nd(m, a, b) = T(n, Nd(m, a, b)) \ .$$

The measure goes down similarly in the remaining recursive applications. The proof system of CL is invoked to check that the measure goes down and if it cannot prove this unassisted it generates proof obligations for the user to satisfy. The checked functions are exactly the unary primitive recursive functions and they can be used in theorems asserting their properties.

The compiler for CL evaluates a definition from its clausal definition. This gives the user the full control of the efficiency of computation. For instance, the Fibonacci function can be computed from its accumulator version $Fiba$ such that $Fib(n) = Fiba(n, 1, 0)$ where

$Fiba(0, a, b) = a$
$Fiba(n + 1, a, b) = Fiba(n, a + b, a)$.

This definition is *tail recursive* and computes with the efficiency of a *while*-loop.

In summarizing this short exposition of the programming language for CL we note that the language contains no reserved words and its clauses can be directly read as logical formulas expressing the properties of defined objects.

4 The Proof System

The proof system of CL is not an automatic theorem prover but rather an intelligent proof checker. It takes care of the mechanical (i.e. decidable) parts of theorem proving and leaves the decisions which are either undecidable or else deemed to be too expensive to the user. Before we elaborate on this we sketch the CL proof of the idempotency of the list reversal function:

$$Rev\ Rev(x) = x \ . \qquad (*_1)$$

The user initiates the proof by specifying *pair* induction on x. The *rule* of pair induction is in general

$$\frac{\phi(0) \qquad \phi(v) \wedge \phi(w) \rightarrow \phi(v,w)}{\phi(x)}$$

for an arbitrary Π_1-formula $\phi(x) \equiv \forall \overline{y} \phi_1(x, \overline{y})$. Pair induction reduces to complete (mathematical) induction because we have $v < v, w$ and $w < v, w$.

In our case the induction formula is the *open* formula $(*_1)$. The *base* case, when $x = 0$, is proved by the CL-proof system automatically as $Rev\ Rev(0) \stackrel{def}{=} Rev(0) \stackrel{def}{=} 0$ because the system automatically unfolds the clausal definitions when it can select an applicable clause uniquely. In the inductive case, when $x = v, w$, the system opens the inner definition

$$Rev\ Rev(v,w) \stackrel{def}{=} Rev(Rev(w) \oplus (v,0))$$

and displays a new goal to prove: $Rev(Rev(w) \oplus (v,0)) = v, w$. The user will now realize that he needs a lemma

$$Rev(x \oplus y) = Rev(y) \oplus Rev(x) \qquad (*_2)$$

correlating concatenation with list reversal. He types the lemma in front of the conjecture $(*_1)$, leaves it unproved for the time being, and tells the CL proof system to use the lemma. The system instantiates the lemma automatically by taking w for x and $(v,0)$ for y and continues:

$$Rev(Rev(w) \oplus (v,0)) \stackrel{*_2}{=} Rev(v,0) \oplus Rev\ Rev(w) \stackrel{def}{=} (v,0) \oplus Rev\ Rev(w) \stackrel{IH}{=} (v,0) \oplus w \stackrel{def}{=} v, w\ .$$

Theorem $(*_1)$ is now conditionally proved and it remains to prove the lemma $(*_2)$ which is done automatically by the system after the user specifies pair induction on x.

The proof system of CL automatically generates new induction rules from suitable predicates. For instance, the rule of *structural induction* on binary trees is automatically generated from the definition of the predicate Bt:

$$\frac{\phi(E) \qquad \phi(a) \wedge \phi(b) \rightarrow \phi(Nd(n,a,b))}{Bt(x) \rightarrow \phi(x)}\ .$$

The rule is sound because Bt is the minimal predicate satisfying its clauses and so it must be included in the property $\phi(x)$ also satisfying the clauses as can be seen from the premises.

The choice of the induction rule or the use of a lemma is in general undecidable and so the proof system of CL does not attempt to do it automatically. For the same reason the system does not automatically instantiate universally quantified formulas such as the used lemmas or inductive hypotheses for Π_1-formulas. Neither does the system automatically split the goals. It leaves the decision to split to the user on the assumption that indiscriminate splitting can lead to the exponential explosion of goals. The proof system, however, in most cases automatically unfolds the clausal definitions and applies the pairing axioms, the properties of linear arithmetic, and the properties of list concatenation.

We can (for the time being) prove in CL only open formulas. The strength of the proof system of CL is characterized as that of $I\Sigma_1$-arithmetic. The last is a fragment of the standard first-order Peano arithmetic where the induction axioms are restricted to Σ_1-formulas (at most existential unbounded quantifiers in the prenex). We can show that every theorem of CL is provable in a recursive extension of $I\Sigma_1$-arithmetic. On the other hand, for every Π_2-formula $\forall x \exists y\, \phi(x)$ the formula is a theorem of $I\Sigma_1$-arithmetic iff we can define a function f in CL and prove that the function *witnesses* the formula ϕ, i.e. that the open formula $\phi(x, f(x))$ is a theorem of CL.

$I\Sigma_1$-arithmetic is that fragment of Peano arithmetic whose *provably recursive* functions (i.e. functions like f above) are exactly the primitive recursive functions. This corresponds to the fact that CL defines only primitive recursive functions.

A typical question computer scientists ask about CL is: Why only primitive recursive functions, why not Ackermann? To that we have two answers. The first one is based on computational complexity and we point out that the primitive recursive functions include as its rather weak proper subclass the elementary functions which are a proper superclass of functions computable in polynomial space (and also time). Non-elementary functions are not feasible to compute so we might as well stop with the primitive recursive functions. This is because that the latter are easier to characterize than the elementary functions by means of 'ptogrammer-friendly' schemas of recursion with measure. The second answer is based on the incompleteness theorem of Gödel. The class of functions we can reason about is given by the strength of the formal system we use. No formal arithmetic allows the introduction of all recursive functions, so it seems to be very natural to stop at the minimal fragment $I\Sigma_1$ which, of course, cannot define the Ackermann function.

5 Specifications

From the logical point of view definitions of functions, predicates, and constants extend the base theory by the addition of new symbols and new axioms (i.e. the clauses). Such extensions are called in formal arithmetic *recursive*. A recursive extension uniquely *expands* the standard model. Extension by a suitable predicate, such as the predicate Bt of being a binary tree, also adds an infinite number of new induction rules generated by the predicate. These rules are always sound in the extended theory.

In order to be able to specify programs and abstract data types CL permits also non-recursive extensions. This is best illustrated with the standard example of stacks. Let us assume that the current extension is a theory S. The extension to a new theory T of stacks adds six new symbols: a constant Es (empty stack); predicate symbols El (elements of stacks) and St (stacks); and function symbols $Push$, Pop, and Sz (size of stacks). Adding these symbols without any axioms is clearly a consistent extension. The first added axioms are

$$Sz(Es) = 0 \tag{1}$$

$$El(x) \to Sz\,Push(x,s) = Sz(s) + 1 \tag{2}$$

$$El(x_1) \wedge El(x_2) \wedge Push(x_1,s_1) = Push(x_2,s_2) \to x_1 = x_2 \wedge s_1 = s_2 \ . \tag{3}$$

It is possible in general (though not in this case) that such an extension makes the theory T inconsistent. Next axioms come from a recursive extension with the measure Sz:

$St(Es)$
$St\,Push(x,s) \leftarrow El(x) \wedge St(s)$.

By recursively defining the property $St(s)$ of s being a stack of elements satisfying El we have effectively said that Es and $Push$ are constructors. This can be seen from the fact that the definition of St automatically adds a rule of structural induction on stacks

$$\frac{\phi(Es) \qquad El(x) \wedge \phi(s) \to \phi(Push(x,s))}{St(s) \to \phi(s)}$$

for every Π_1-formula $\phi(s)$. Such an extension by infinitely many rules cannot make consistent theory inconsistent. We also add an axiom constraining the operation Pop:

$$El(x) \wedge St(s) \to Pop\,Push(x,s) = x, s \tag{4}$$

which asserts the basic property of stacks: last in first out. This ends the extension to the theory T of stacks over El.

The situation now reverts to the point just before we have added the six new symbols, i.e. we are again in the theory S. We may now keep extending S by recursive extensions and perhaps prove some theorems until we have a theory S_1 where we decide to *implement* the abstract data type of stacks. Implementation (of an abstract data type) is a standard term in the specification circles.

The same is achieved in CL by purely logical means. We simply *interprete* in S_1 the symbols of T with the exception of the symbol St because it has a unique definition from El, Es, and $Push$. We might have decided to interprete the stacks in the simplest possible way by lists of natural numbers. For that we might have defined in S_1 the five symbols as follows:

$$
\begin{aligned}
&Es = 0\\
&El(x)\\
&Push(x) = x\\
&Pop(x) = x\\
&Sz(0) = 0\\
&Sz(v, w) = Sz(w) + 1 \ .
\end{aligned}
$$

We must now prove the axioms (1) through (4) of T. It should be obvious that they can be easily proved and we thus know that the theory T is consistent.

The reader will note that instead of types (sorts) we use predicate symbols in CL. There is no need for any higher order constructs because the parameterization of specifications is done by interpretations within extensions (specifications) similarly as in Larch [8]. The reader will also note that the specification apparatus is absolutely minimal and consists only of extensions with new symbols and axioms as well as of interpretations.

References

1. R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, 1979.
2. R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Proceedings of 5th Internat. Joint Conf. on Artificial Intelligence*, pages 1045–1058, 1977.
3. T. Coquand and G. P. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *Proceedings of EUROCAL 85.* Springer Verlag, 1985.
4. R. L. Constable et al. *Implementing Mathematics with the Nuprl.* Prentice-Hall, 1986.
5. K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings of POPL'85*, pages 52–66, 1985.
6. M. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer, 1988.
7. M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS.* Springer Verlag, 1979.
8. J. V. Guttag, J. J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification.* Springer Verlag, 1993.
9. P. Hajek, P. Pudlak. *Metamathematics of First Order Arithmetic.* Springer, 1993.
10. W. McCune. OTTER 2.0 users guide. Technical report ANL-90/9, Argonne National Laboratory, 1990.
11. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Proceedings of CADE-11*, number 607 in LNAI, pages 748–752. Springer Verlag, 1992.
12. M. Wirsing. Structured algebraic specifications. In B. Robinet, editor, *Proceedings of AFCET'82*, pages 93–108, 1982.