From Declarative to Imperative Proof-Carrying Programs

Ján Kľuka, Paul J. Voda
Institute of Informatics, Comenius University Bratislava,
Mlynská dolina, 842 48 Bratislava, Slovakia,
email: {kluka,voda}@fmph.uniba.sk

Abstract

We propose a methodology for the translation of high-level declarative programs into low-level declarative programs. The latter are such that they can be immediately transformed to imperative programs. The translation automatically generates correctness properties connecting the results of the high-level and low-level programs. Moreover, the correctness properties should be automatically provable. This transfers any previously proved properties of the high-level programs to the proved properties of the low-level programs. The low-level proofs can be then checked by a (dumb) proof checker. In this sense the generated programs carry their own correctness proofs.

The low-level programs work explicitly with the store and it is possible to specify its reuse which means that, for instance, a declarative array obtained by a modification of an argument array can reuse the store of the argument array. We outline the methods of extensible, programmer specified, representation of high-level data types in the store.

We argue that the naive use of the imperative style of working with arrays in a declarative setting leads to unmanageable proofs of correctness properties. We propose a somewhat surprising style of verification of high-level array programs where we deteriorate their efficiency in order to facilitate easier correctness proofs. The efficiency is restored by the translation.

1 Introduction

With compilers for high-level languages there is almost no need to program in assembly languages. Some of the modern language processors (usually for declarative languages) come with their own intelligent proof assistants where one can prove properties of programs. Such integrated systems are so complex that the present day technology does not permit their verification. As a consequence, one has no guarantee that the verified programs remain correct when translated to low-level ones. Thus there is a genuine need (at least for the mission critical programs) to transfer the proofs of properties of high-level programs to the proofs of properties of the translated low-level programs. The low-level proofs can be then verified by a simple (dumb) proof checker.

The first of the two results of this paper is the proposal of a methodology for the transfer of proofs. We have used in teaching for seven years now a high-level declarative programming language CL (Clausal Language) whose formal theory is the simplest non-trivial logical theory: *Peano Arithmetic* (PA). Our implementation permits the verification of CL programs with the help of an intelligent proof assistant supervising the construction of proofs in PA. CL is implemented as a server communicating with a web browser supporting the mathematical markup language (MathML). For an online demo see [3].

We will define a sublanguage of CL, called *Peano Pascal* (PP), which, while still declarative, is very close to low-level assembly language programs. PP programs work with, say 32-bit, words holding small numbers and pointers. PP programs directly access and/or modify memory viewed as partial functions over words.

We will outline a method for the automatic translation of a high-level CL function f to a low-level PP function f' which can be then directly transformed to a machine level or a C program. The translator formulates in PA a correctness property tying together the values of f and f'. Just as we do not wish to program at the level of PP, we do not wish to prove the correctness property by hand. Based on our long experience with the verification of CL functions in PA and of the second author's experience with *proof theory* (a branch of mathematical logic concerned with the (possibly automatic) transformation of proofs) we have a reason to believe that the proof of the correctness property can be automatically constructed by the translator. The correctness property, now a theorem of PA, automatically transfers the properties of f to the properties of f'. This is the subject of the PhD thesis of the first author.

Declarative PP functions transformed to imperative C programs permit in-place modification of memory. Thus list processing programs can be transformed into fast array processing programs enjoying the constant time indexing and savings of pairing operations (*cons*). The second result of the paper is a proposal for the methodology for the verification of such programs. We will argue that the 'naive' mimicking in high-level programs of array indexing/modification via corresponding list operations leads to programs which are quite hard to verify. Analysis of this phenomenon shows that the culprit is the *non-locality* of array processing. Declarative programs—in contrast to imperative ones with their side effects—restrict the access only to the arguments of functions. Properties of such functions are thus quite easy to express within PA. Naive use of arrays reintroduces non-locality because array programs typically recursively operate on their proper segments while entire arrays must be passed as arguments. Formulas of PA expressing properties of naive array programs must then use indexing and quantification to specify which portions of argument arrays do not change.

The closest work related to ours is by J. Reynolds and P. O'Hearn [4] who are concerned with the verification of low-level imperative programs. We have noticed the non-locality of naive array processing independently of their analysis of non-locality of accesses to the store as the major stumbling block on the way to program verification. The two are identical because the full access to arrays leads to the same problems as the full access to the store. We have adapted their store partitioning operation $\star$ within our framework of declarative programming, but not their *spatial conjunction* because this is usable only within the Hoare-style methodology of verification of imperative programs.

We think that our declarative approach is simpler because all three components: CL, PP, and the assertion language are subsets of the same language of PA. Corresponding components in [4] (recursive, command, and assertion languages) are three distinct formal systems. Also, [4] uses a *tagged* infinite model of the store while we use an untagged finite one which is closer to how the computers work. We permit the mixing of small numbers and pointers which leads to our type N of (unlimited) natural numbers (see Par. 3.5) not directly expressible in the formalism of [4]. Also their 'exact' representation of data-types such as trees precludes the sharing of subtrees which is typical of declarative programs.

Also related is the extensive research (see, for instance [1, 5, 6, 9]) in the analysis of declarative programs with the intention of reuse of storage. Most of such work calls for the automatic analysis of programs, while our experience is that one needs a careful planning of the reuse of memory and the translator needs explicit instructions for this. This can be seen in our examples of Quick (Par. 4.6) and Merge (Par. 4.7) sorts. Incidentally, the system CL is implemented in the second author's declarative programming language Trilogy which specifies the reuse of store in a similar (if somewhat simpler) way as reported here.

2 Quick Overview of CL

2.1 Natural numbers as the domain of CL

Theory of computability deals with computable functions over the domain of natural numbers ($\mathbb{N}$). A formal theory of natural numbers—Peano arithmetic—is the basic first-order theory of mathematical logic and every computer scientist is familiar with it.

A *pairing* function, say the Cantor's diagonal function, $x, y = (x+y)\cdot(x+y+1) \div 2 + x + 1$, has properties similar to the *cons* of LISP (0 plays the role of *nil*). Every natural number x is either 0 or it can be uniquely written as a *list* $x = x_1, \ldots, x_n, 0$ where we let pairing associate to the right. Data structures, say binary trees (with labels from $\mathbb{N}$), can be constructed with the help of *constructors* (functions over $\mathbb{N}$): $\bullet = 0$ and $\dfrac{n}{l \mid r} = n, l, r$ (see Par. 4.8). We can thus get the coding comfort of S-expressions of LISP within the domain of natural numbers as formalized in PA. By means of *extensions by definitions* of PA (see [10]) we can conservatively introduce into PA computable functions and predicates whose definitions can act as programs.

For these reasons we have designed the programming language CL on top of PA. In order to obtain modularity (parameterization by functions and predicates) we are preparing a new version of CL which will be based on a subsystem of the second-order arithmetic called ACA_0, which is conservative over PA (see [12]).

2.2 Basic CL operations

For $i = 0, 1$ we introduce into PA the *binary successors* $S_i(x) = 2\cdot x + i$ abbreviated as $x\underline{0}$ and $x\underline{1}$ respectively. Every x (in $\mathbb{N}$) is of one of the two forms $x = y\underline{0}$ or $x = y\underline{1}$ for a unique y, said to be obtained by *pattern matching*. This can be used in *binary discriminations* (case analysis) to define functions:

$$f(x) = \textbf{case } x = y\underline{0} \Rightarrow \ldots x \ldots y \ldots \quad \textbf{case } x = y\underline{1} \Rightarrow \ldots x \ldots y \ldots$$

possibly by a fast *recursion on notation*.

Similarly, *pair discrimination* uses the fact that for any x exactly one of $x = 0$ or $x = v, w$ holds for uniquely determined $v < x$ and $w < x$. Using pair discrimination, the *list concatenation* function $\oplus$ is introduced into PA to satisfy $0 \oplus y = y$ and $(v, w) \oplus y = v, (w \oplus y)$. We also introduce the *list length* function s.t. $L(0) = 0$ and $L((v, w)) = L(w) + 1$, the *singleton* function $[x] = x, 0$ and the *list reversal* function s.t. $Rev(0) = 0$ and $Rev((v, w)) = Rev(w) \oplus [v]$.

We will also need the *list splitting* discrimination where for any a and any $n > 0$ exactly one of the two alternatives holds: $L(a) < n$ or $a = b \oplus c \wedge L(b) = n$ for uniquely determined b and c and also the *singleton* discrimination where for any a such that $L(a) = 1$ we have $a = [x]$ for a unique x.

Note that we can view the uniquely determined output variables in discriminations as determined by pattern matching.

$$
\begin{array}{l}
Part(c) = \textbf{case } c = 0 \Rightarrow 0 \\
\qquad\qquad \textbf{case } c = d_1 \oplus d \wedge L(d_1) = 1 \Rightarrow \textbf{let } [p] = d_1 \textbf{ in} \\
\qquad\qquad\quad \textbf{case } a = 0 \Rightarrow 0, [p], 0 \\
\qquad\qquad\quad \textbf{case } a = d \oplus d_2 \wedge L(d) + 1 = L(a) \Rightarrow \textbf{let } [v] = d_2 \textbf{ in} \\
\qquad\qquad\qquad \textbf{case } p \le v \Rightarrow \\
\qquad\qquad\qquad\quad \textbf{let } l, r, g = Part([p] \oplus a) \textbf{ in } l, r, (g \oplus [v]) \\
\qquad\qquad\qquad \textbf{case } p > v \Rightarrow \\
\qquad\qquad\qquad\quad \textbf{case } d = 0 \Rightarrow [v], [p], 0 \\
\qquad\qquad\qquad\quad \textbf{case } d = b_1 \oplus b \Rightarrow \textbf{let } [w] = b_1 \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{let } l, r, g = Part([p] \oplus b \oplus [w]) \textbf{ in } ([v] \oplus l), r, g
\end{array}
$$

Figure 1. The case form of the clausal definition of *Part* from Fig. 6

2.3 Clausal definitions

Every CL definition can be visualized in the form of a nested case expression with cases corresponding to discriminations. For example, in Fig. 1 the reader will recognize the list splitting, singleton, and the *dichotomy* ($p \le v$ vs. $p > v$) discriminations. Also the **let** construct can be visualized as a discrimination with one alternative whose sole purpose is to assign values to local variables (possibly by pattern matching). We should view the expression in Fig. 1 as a tree and read off its branches. Each branch, after applying to it some equivalence-preserving simplifications which are provable in PA, becomes a clause of a clausal definition of the function *Part* given in Fig. 6. The clause for the first branch is $Part(c) = 0 \leftarrow c = 0$, which can be simplified to $Part(0) = 0$, and it is omitted by *default* from Fig. 6 because it yields the 'dummy' value 0 as *Part* will always be called with $c \neq 0$.

The reason why we use in CL clauses and not case expressions is that the clauses are fully expressible in the language of PA and so CL has no reserved words and no special syntax beyond that of first-order logic. Moreover, a clause is an implication (written in a converse form), and an axiom stating properties of the defined function or predicate. See [11] for the description of how the clauses are equivalently reduced to the required form of extension of PA by definitions.

Every recursive CL definition, say of $f(\overline{x})$, must be *regular*, i.e., PA must admit a *measure* function $m(\overline{x})$ such that every recursive call $f(\overline{\tau})$ decreases the measure of arguments: $m(\overline{\tau}) < m(\overline{x})$ provably in PA.

For the automatic transfer of proofs to the level of PP we crucially need *conditional* clausal definitions (see [11] for more details). For instance, assume that we have defined the two place *list indexing* function $a[i]$ yielding the i-th element of the list a. We can then define the three-place *list modification* function $a[i{:=}v]$ yielding a list satisfying:

$$\begin{aligned} i < L(a) \rightarrow (i \neq j < L(a) \rightarrow a[i{:=}v][j] = a[j]) \wedge \\ (i = j \rightarrow a[i{:=}v][j] = v) \ . \end{aligned} \tag{1}$$

Both functions are clausally defined under the condition $i < L(a)$ that the index i is within the list a. The definition of the list modification function is:

$$\begin{aligned} &a[0{:=}v] = v, a && \leftarrow a = x, b && \leftarrow 0 < L(a) \\ &a[i+1{:=}v] = b[i{:=}v] && \leftarrow a = x, b \ . && \leftarrow i+1 < L(a) \end{aligned}$$

Every call of $a[i{:=}v]$ and of $a[i]$ (recursive or from another clausal definition) will result in a proof obligation assuring that the condition is met.

3 Peano Pascal

3.1 Words and simple types formed from words

We can view PP functions as typed by *simple types* σ each of which has a *length* $L(\sigma)$. We define the *word* predicate, say, as $w\colon W \leftrightarrow w < 2^{32}$ with the length $L(W) = 1$.

We then form simple types by cartesian products and disjoint unions starting from W. For each simple type σ we introduce into PA a unary predicate $x\colon \sigma$. If the predicate symbols for σ, τ have been introduced, then we can define for the type $\sigma \times \tau$ of length $L(\sigma) + L(\tau)$ the predicate

$$x\colon \sigma \times \tau \leftrightarrow \exists y \exists z (x = y, z \wedge y\colon \sigma \wedge z\colon \tau) \ .$$

We abbreviate $\sigma \times (\tau \times \rho)$ to $\sigma \times \tau \times \rho$, $\sigma \times \sigma$ to σ^2, and similarly for higher powers.

If the predicates for $\tau_1, \ldots, \tau_n$ have been introduced into PA we can introduce a predicate of their disjoint union: $\bigcup\{\tau_1, \ldots, \tau_n\}$ of length $\max(L(\tau_1), \ldots, L(\tau_n)) + 1$ to satisfy:

$$x\colon \bigcup\{\tau_1, \ldots, \tau_n\} \leftrightarrow x\colon \{1\} \times \tau_1 \vee \cdots \vee x\colon \{n\} \times \tau_n \ .$$

Here the auxiliary *singleton* predicates (serving as tags) are defined as $x\colon \{i\} \leftrightarrow x = \underline{i}$ where $\underline{i}$ is a *numeral* denoting i, say the term $S^i(0)$. We also permit one or more of τ_i to be $\{0\}$ (with length 1).

3.2 Stores

Store s is a partial function over words ($W \rightarrow W$). Every such function is finite, and it can be represented in PA by a list with elements $(a, v)\colon W^2$ increasingly ordered by a, where a is the argument of the partial function, and v is its value. We leave to the reader the introduction into PA of the *store* predicate $s\colon S$ holding for such lists. We will use the predicate symbol S in cartesian products to form the *extended types*, for instance $W \times S$.

It is straightforward to introduce into PA the *domain* function $dom(s)$ which for $s\colon S$ yields the increasingly ordered list of first components of s. We can easily introduce into PA the basic set-theoretical operations such as $\cup$, $\cap$, or $\subseteq$ operating on ordered lists representing finite sets.

The three-place *store partitioning* relation $s \doteq s_1 \star s_2$ is introduced into PA to hold iff s, s_1, s_2 are in S and such that $dom(s_1) \cup dom(s_2) = s$ and $dom(s_1) \cap dom(s_2) = 0$. For a formula $\phi[x]$ we then abbreviate by $\phi[s_1 \star s_2]$ the formula $\exists x(x \doteq s_1 \star s_2 \wedge \phi[x])$. Thus, for instance, $s \doteq s_1 \star s_2 \star s_3$ abbreviates $\exists x(x \doteq s_2 \star s_3 \wedge s \doteq s_1 \star x)$, and $f(s) = s_1 \star s_2$ abbreviates $\exists x(x \doteq s_1 \star s_2 \wedge f(s) = x)$.

The *empty* store is 0, and the list $s = (p, i_0), (p+1, i_1), \ldots, (p + n - 1, i_{n-1}), 0$ will be abbreviated to $[p \mapsto i_0; i_1; \ldots; i_{n-1}]$ ($n > 0$ must be a constant). Note that for $s\colon S$ we must have $p + n - 1$, i_0, i_1, …, i_{n-1} in W. Such stores s are called *array segments* and they are designated with a pair of words (p, n). We introduce into PA a binary relation $s \risingdotseq v$ saying that *s is an array segment designated by v*, i.e., holding iff $s\colon S$, $v = (p, n)\colon W^2$ for some p, n, and $dom(m) = p, p{+}1, \ldots, p{+}n{-}1, 0$. Note that we have $0 \risingdotseq p, 0$. We also define the *segment length* function $L(t)$ s.t. $L((p, n)) = n$.

We introduce into PA by conditional clausal definitions the *store indexing* $s[p]$, and *modification* $s[p{:=}w]$ functions. They are similar to the corresponding list functions except that they do not index by position within s but by arguments p in the elements (p, v) of s.

3.3 Functions of Peano Pascal

PP functions are a subclass of CL functions. PP functions, by being declarative, afford proofs in PA of their properties. On the other hand, PP functions are extremely low-level because they are permitted only to access their arguments (by pairing), to access/modify/allocate the store, and to call previously defined PP functions. The arguments of calls are constructed by pairing and by the basic arithmetical operations ($+$, $\cdot$, …). The only discriminations are numeric discriminations, for instance, dichotomy, and the pair discrimination. The latter only with single alternatives $x = v, w$ in contexts where we know that $x\colon W^2$.

An $(n+1)$-ary PP-function $f'(\overline{x})$ accessing the store is conditionally defined with a condition (provably) implying $\overline{x}\colon \sigma \times S$ where σ is a simple type of length n. The type $\sigma \times S$ is called the *input type* of f'. Proof obligations thus generated will guarantee that the arithmetical operations do not overflow the words.

The function f' must provably satisfy $\bar{x}\colon \sigma \times S \to f'(\bar{x},s)\colon \tau$ where τ, called the *output type*, is either $\tau_1 \times S$ or $\bigcup\{\tau_1 \times S, \{0\}\}$ for a simple type τ_1. The second form of output typing is used to signal the *heap exhausted* condition when f' yields $2,0$.

PP functions not accessing the store are typed similarly but without the type S. An example is the *heap allocation* function $New'\colon W \times W^2 \to \bigcup\{W \times W^2, \{0\}\}$ which allocates k words into p from the heap h designated by an array segment:

$$New'(k,h) = 1, p, (p+k, n \dot{-} k) \leftarrow h = p,n \wedge k < n \leftarrow k,h\colon W \times W^2$$
$$New'(k,h) = 2,0 \leftarrow h = p,n \wedge k \geq n \leftarrow k,h\colon W \times W^2$$

The function does not access the store because it operates on the heap designator only.

The above (declarative) PP function f' is on such a low level that it can be directly transformed into an imperative machine program, or for a better visualization, into a C function F. In the latter case F will have n arguments of type *long unsigned*, and $L(\tau)$ *var* arguments, i.e., arguments of type *long unsigned*$*$. The store is omitted from the definition of F.

The arithmetic in F is done by the *long unsigned* arithmetic, indexings of store $s[p] = x$ in f' are replaced by assignments $x = *((\mathit{long\ unsigned}*)p)$, modifications of store $s[p{:=}v] = s_1$ in f' by indirect assignments $*((\mathit{long\ unsigned}*)p) = v$, and heap allocations $New'(k,h) = p,h'$ in f' by calls on a C function invoking *malloc*.

No heap is needed for the pairing operations on arguments of F (or of functions called from it) because each input and output argument of f' (or of the called function) maps directly to an argument of F (or of the called function).

3.4 Representation of high-level CL values in the store

We can represent in store by a *PP-type* T a subset of high level CL objects (of natural numbers that is) given by a unary predicate P. This is done by designing a three-place *representation* predicate $x \sim_s^T p$ satisfying:

$$x \sim_s^T p \wedge s \subseteq s_1 \to P(x) \wedge s\colon S \wedge p\colon \tau \wedge x \sim_{s_1}^T p\ .$$

Here τ is a simple type, most often W. We say that *p represents the value x in the store s*. Note that this guarantees *monotonicity* expressing the fact that a larger store (s_1) does not change the representation of x.

With every PP-type T we associate a set of basic high-level CL operations (including discriminations) and type them with the help of T. We then program (by hand) corresponding PP operations, and correlate the two sets by properties proved by hand (see Paragraphs 3.5, 4.1, and 4.4). The pairs of operations and their properties will be used by the CL-to-PP translator.

3.5 Representation of natural numbers

If our program works with small numbers, we can represent them in the store directly by defining a PP-type W with the representation predicate introduced as $x \sim_s^W n \leftrightarrow n\colon W \wedge x = n$. However, the design of high-level CL functions with arguments to be mapped into the PP-type W will not be very pleasant. We will have to satisfy possibly many proof obligations generated for such functions in order to prevent overflows.

$$p \sim_s^N p\underline{11} \quad \leftarrow p\underline{11}, s\colon W \times S$$
$$n\underline{0} \sim_s^N p\underline{01} \quad \leftarrow p\underline{01}, s\colon W \times S \wedge n > 0 \wedge s \doteq [p\underline{00} \mapsto q] \star s_1 \wedge n \sim_{s_1}^N q$$
$$n\underline{1} \sim_s^N p\underline{10} \quad \leftarrow p\underline{10}, s\colon W \times S \wedge s \doteq [p\underline{00} \mapsto q] \star s_1 \wedge n \sim_{s_1}^N q$$
$$n,m \sim_s^N p\underline{00} \leftarrow p\underline{00}, s\colon W \times S \wedge s \doteq [p\underline{00} \mapsto q;r] \star s_1 \wedge n \sim_{s_1}^N q \wedge m \sim_{s_1}^N r$$

Figure 2. Mixed mode representation of natural numbers

Alternatively, one can decide to work with natural numbers of 'unlimited' precision (subject to heap exhaustion) and represent natural numbers by lists of words. However, every CL value is a natural number, and we can perform in the high-level CL programs arbitrary pairing and/or arithmetical operations on them. Thus it is of advantage to use the *mixed mode* representation of natural numbers by a PP-type N whose representation predicate is given by the clausal definition in Fig. 2. Note that if $x \sim_s^N p$ holds, then the two least significant bits of p determine whether p directly holds a small number $<2^{30}$ (the first clause), whether p points to a *cons*-cell (the fourth clause), or whether p represents one of the two binary successors.

Basic CL operations (which by [11] are sufficient for the efficient programming of all computable CL functions) are typed by N as follows: $0\colon N$, $S_i\colon N \to N$, and $P = \lambda xy.(x,y)\colon N^2 \to N$. Discriminations are typed by $N \to \bigcup\{\{0\},\{0\}\}$ (test on zero), $N \to \bigcup\{N,N\}$ (binary discr.), and $N \to \bigcup\{\{0\}, N \times N\}$ (pair discr.).

The corresponding PP operations are typed as $0\colon W$, $S_i'\colon W \times W^2 \times S \to \bigcup\{W \times W^2 \times S, \{0\}\}$, and $P'\colon W \times W \times W^2 \times S \to \bigcup\{W \times W^2 \times S, \{0\}\}$. We give here just the clausal definition of P':

$$P'(p,q,h,s) = 1, r, h', s' \leftarrow New'(2,h) = 1, r, h' \wedge s[r{:=}p][r+1{:=}q] = s' \quad \leftarrow p,q,h,s\colon W \times W \times W^2 \times S$$
$$P'(p,q,h,s) = 2,0 \quad \leftarrow New'(2,h) = 2,0 \leftarrow p,q,h,s\colon W \times W \times W^2 \times S$$

Both pairing operations are connected together with the partial correctness property proved in PA:

$$n \sim_s^N p \wedge m \sim_s^N q \wedge s_1 \mathrel{\bumpeq} h \wedge P'(p,q,h,s \star s_1) = 1, r, h', s' \to \exists s_1' \exists s_2 (s' = s \star s_2 \star s_1' \wedge (n,m) \sim_{s \star s_2}^N r \wedge s_1' \mathrel{\bumpeq} h')\ .$$

S_i' will perform optimization when the result can be represented as a small number (this corresponds to, say, $S_i(128)$).

The basic PP discrimination functions for N are slightly more complicated. For instance, the function Db' corresponding to binary discrimination is typed as $Db'\colon W \times W^2 \times S \to \bigcup\{W \times W^2 \times S, W \times W^2 \times S, \{0\}\}$. If $Db'(p,h,s) = 1, h', s'$ then the call may perform conversion (by computing the pairing function) if, for instance, p points to a a *cons*-cell (this corresponds to, say, the high-level case $a,b = y\underline{0}$). The computation may allocate from the heap. The returned tag 2 corresponds to the case when p represents an odd number and the tag 3 signals the heap exhausted condition.

The function Dp' corresponding to the pair discrimination is typed similarly because a numeric argument, say $x+3$, can be broken down into a form v,w only by computing of both projection functions.

Such computations are time consuming but, fortunately, they are rather pathological since CL programs normally observe a type discipline and do not perform arithmetic on *cons*ed arguments or use pair discrimination with the result of an arithmetical operation.

3.6 CL to PP translation

Translation of an n-ary high-level CL function f into a PP-function f' is triggered by the typing of f with PP-types as $T_1 \times \cdots \times T_n \to T$. The translator checks whether the clauses of f are correctly typed, determines the typing of f' by simple types, and uses the PP operations/discriminations corresponding to the high-level ones for the construction of f' (see Fig. 3).

We can visualize the translation of f performed by the current version of CL (which does not work with PP) by the typing $f\colon N^n \to N$. The translated PP-function will be typed as $f'\colon W^n \times W^2 \times S \to \bigcup\{N \times N^2 \times S, \{0\}\}$, the condition on its definition will be:

$$\exists s_0 \exists s_1 \exists \overline{x}(s \doteq s_0 \star s_1 \wedge x_1 \sim^N_{s_0} p_1 \wedge \cdots \wedge x_n \sim^N_{s_0} p_n \wedge s_1 \mathrel{\hat{=}} h) \ ,$$

and the *partial* correctness property will be:

$$\begin{aligned} s \doteq s_0 \star s_1 \wedge x_1 \sim^N_{s_0} p_1 \wedge \cdots \wedge x_n \sim^N_{s_0} p_n \wedge \\ s_1 \mathrel{\hat{=}} h \wedge f'(\overline{p},h,s) = 1,q,h',s' \to \\ \exists s'_1 \exists s_2(s' \doteq s_0 \star s_2 \star s'_1 \wedge s'_1 \mathrel{\hat{=}} h' \wedge f(\overline{x}) \sim^N_{s_0 \star s_2} q) \end{aligned}$$

Note that the store s_2 is allocated from the heap h. The correctness assures that the properties proved for f carry over to the value represented in the store s' by the result pointer q. Also note that by the monotonicity of representation, the values of the input arguments $\overline{x}$ are not destroyed in final store s'. We can, namely, derive $x_i \sim^N_{s_0 \star s_2} p_i$ for each argument x_i.

Unless we are willing to find a bound on the size of $f(\overline{x})$ (which is always possible with a primitive recursive f) and prove it by hand, we must expect the *heap exhausted* condition. This is the only reason for partial correctness because the termination of f' is assured from the termination of f, the arithmetic on words will never overflow, and no dangling pointer will be dereferenced. This is assured by proof obligations stemming from the conditions on PP functions.

4 Representation of Lists by Arrays

4.1 Arrays of fixed size

A list of a not changing length n can be represented in the store as an array of a fixed size, i.e., by n consecutive locations. Assume that we have previously defined a PP-type T and that we are given a constant $n\colon W$. The PP-type $A(n,T)$ (recall that this is just a name) of arrays containing n values represented by a type T is given by the representation predicate $a \sim^{A(n,T)}_s p$. For its definition we introduce first a three-place predicate $a \simeq^{A(T)}_s s_1$ saying that the *elements of the list a represented in s are contained in the array segment s_1*:

$$\begin{aligned} a \simeq^{A(T)}_s s_1 \leftrightarrow \exists s_0(s_0 \doteq s \star s_1) \wedge \\ \exists p(s_1 \mathrel{\hat{=}} (p, L(a)) \wedge \forall i{<}L(a)(a[i] \sim^T_s s_1[p+i])) \ . \end{aligned}$$

The array representation predicate has then the following definition:

$$a \sim^{A(n,T)}_s p \leftrightarrow \exists s_1 \exists s_2(s \doteq s_1 \star s_2 \wedge a \simeq^{A(T)}_{s_1} s_2 \mathrel{\hat{=}} (p,\underline{n})) \ .$$

There are three high-level CL operations typed with $A(n,t)$, namely allocation $New(\underline{n})\colon A(n,T)$, indexing: $\lambda ai.a[i]\colon A(n,t) \times W \to T$ and modification:

$$\lambda aiv.a[i{:=}v]\colon (a\colon A(n,t) \times W \times T \to b\colon A(n.t)) \qquad \text{with } a \simeq b$$

Here $New(k)$ yields a list of length k containing zeroes. The typing of the list modification function specifies that the translated PP program should overlay the array segment representing the argument a with the result b, i.e., that the segments should be the same. Recall that there is no semantics to typing, it serves just as an instruction to the translator. Applications of the three operations are translated in that order to the calls of the PP allocation New', of store indexing, and of store modification functions. The list and store indexing operations are connected by the following PA provable properties:

$$a \simeq^{A(T)}_{s_1} s_2 \mathrel{\hat{=}} (p,k) \wedge i < k \to a[i] \sim^T_{s_1} s_2[p+i]$$

$$\begin{aligned} a \simeq^{A(T)}_{s_1} s_2 \mathrel{\hat{=}} (p,k) \wedge i < k \wedge v \sim^T_{s_1} q \to \\ a[i{:=}v] \simeq^{A(T)}_{s_1} s_2[p+i{:=}q] \mathrel{\hat{=}} (p,k) \ . \end{aligned}$$

The properties will guarantee that the CL to PP translation will be correct provided that the CL programs posses a *single reference* property, which means that after an array modification operation $a[i{:=}v] = a_1$, we do not access in the rest of the clause the 'old' version a sharing the same arrays segments in the translated program.

$$\begin{aligned} &Revv_1(a,i,j) = a && \leftarrow i+2 \geq j \\ & && \leftarrow i < j \leq \underline{n} = L(a) \\ &Revv_1(a,i,j) = Revv_1(a_2, i+1, j \dot{-} 1) \leftarrow i+2 < j \wedge \\ &\quad a[j \dot{-} 1] = x \wedge a[i] = y \wedge a[j \dot{-} 1{:=}y] = a_1 \wedge a_1[i{:=}x] = a_2 \\ & && \leftarrow i < j \leq \underline{n} = L(a) \end{aligned}$$

$$Revv(a) = Revv_1(a,0,L(a)) \qquad \leftarrow \underline{n} = L(a)$$

$$\begin{aligned} &Revv'_1(p,i,j,s) = s \leftarrow i+2 \geq j && \leftarrow \exists a\, a \sim^{A(n,T)}_s p \wedge i < j \leq \underline{n} \\ &Revv'_1(p,i,j,s) = Revv'_1(p,i+1,j \dot{-} 1, s_2) \leftarrow i+2 < j \wedge \\ &\quad s[p+j \dot{-} 1] = x \wedge s[p+i] = y \wedge s[p+j \dot{-} 1{:=}y] = s_1 \wedge \\ &\quad s_1[p+i{:=}x] = s_2 && \leftarrow \exists a\, a \sim^{A(n,T)}_s p \wedge i < j \leq \underline{n} \end{aligned}$$

$$Revv'(p,s) = Revv'_1(p,0,\underline{n},s) \qquad \leftarrow \exists a\, a \sim^{A(n,T)}_s p$$

Figure 3. Naive high-level and PP implementations of array reversal

4.2 In-place list reversal programmed naively

It is now quite straightforward to program a high-level list reversal function $Revv$ mimicking the reversal of elements of an array in an imperative programming language. Fig. 3 gives this 'naive' program (relative to a fixed T and n) where the auxiliary function $Revv_1$ works from both ends of the list a by successively swapping the elements of a at index positions i and $j-1$. Both functions have attached conditions which will guarantee the automatic proof of the correctness property given below. The condition on $Revv_1$ is satisfied in its initial call from *Revv* and maintained by the recursive calls.

We have written the second clause for $Revv_1$ in an expanded way so the reader can observe the flow of data where the old versions a and a_1 of the argument are not accessed after their modifications. The clause can be equivalently written in a contracted form as follows:

$$Revv_1(a,i,j) = Revv_1(a[j \dot{-} 1{:=}a[i]][i{:=}a[j \dot{-} 1]], i+1, j \dot{-} 1) \\ \leftarrow i+2 < j \ .$$

The translation to PP is triggered by the typing $Revv\colon (a\colon A(n,T) \to b\colon A(n,T))$ with $a \simeq b$, and $Revv_1\colon (a\colon A(n,T) \times W \times W \to b\colon A(n,T))$ with $a \simeq b$.

There is no allocation of store, hence no heap designator, and the typing of, say $Revv'(p,i,j,s) = s_1$, is $W \times W \times W \times S \to S$. Note that because of store sharing there is no need to return a pointer to the result array (it is pointed to by p).

The following (total) correctness condition should be provable automatically:

$$a \simeq_s^{A(T)} s_1 \mathrel{\widehat{\approx}} (p,\underline{n}) \to \\ \exists s_1'(Revv'(p,s\star s_1) = s\star s_1' \wedge Revv(a) \simeq_s^{A(T)} s_1' \mathrel{\widehat{\approx}} (p,\underline{n}))\ .$$

The correctness of $Revv'$ with respect to $Revv$ is much easier to obtain than the high-level correctness property $Revv(a) = Rev(a)$. Its proof is, of course, by hand and it needs a lemma about $Revv_1(a,i,j)$ asserting that provided that $i < j < L(a)$, the call yields a list whose segment between i and j is reversed and the segments between 0 and i as well as between i and $L(a)$ are unchanged. If one expresses this by a formula involving quantifiers and indexing then its proof will involve a lot of arithmetic and will be quite tedious because it has to use the property 2.3(1) leading to repeated case analysis and heavy manipulation of quantifiers.

The reader should realize that *Revv* is a simple array program. The programs presented below lead to quite difficult proofs when programmed in a naive way with the explicit use of array indexing and modification.

4.3 Methodology for the verification of array programs

The conclusion from the previous paragraph is that, although extremely efficient, arrays are inherently imperative. Naive use of arrays in a high-level declarative programming language leads to difficult verification proofs. This is because arrays are very close to stores and programs (both imperative and naive declarative ones) typically work only on their segments. The advantage of declarative programming is that the segments can be passed as arguments of functions and operations on them cannot affect the unmodified segments of arrays.

We will now discuss a methodology for the high-level work with arrays which still enables automatic translation and verification. The methodology is somewhat surprising. One, namely, deteriorates the efficiency of high-level CL programs in order to obtain easier verification proofs. By using list concatenation, the high-level programs will not have to use list indexing/modification. Their efficiency deteriorates because of splitting discriminations, for instance, for extracting the last element of a list. The verification proofs without explicit indexing need almost no arithmetic. The efficiency is restored by the translation to a PP array program.

$$t \oplus' u = p, k+n \leftarrow t = p,k \wedge u = q,n \qquad \leftarrow adj(t,u)$$

$$Singl'(p,(q,1),s) = s[q{:=}p] \\ \leftarrow \exists s_1 \exists s_2 \exists x(s \doteq s_1 \star s_2 \wedge x \sim_{s_1}^{T} p \wedge s_2 \mathrel{\widehat{\approx}} q,1)$$

$$Dconc'(t,n) = 1,0 \qquad \leftarrow t = p,k \wedge k < n \\ \leftarrow t,n\colon W^2 \times W$$

$$Dconc'(t,n) = 2,(p,n),(p+n,k \dot{-} n) \leftarrow t = p,k \wedge k \geq n \\ \leftarrow t,n\colon W^2 \times W$$

$$Dsingl'(t,s) = s[p] \leftarrow t = p,1 \\ \leftarrow \exists s_1 \exists s_2 \exists x(s \doteq s_1 \star s_2 \wedge [x] \simeq_{s_1}^{A(T)} s_2 \mathrel{\widehat{\approx}} t)$$

Figure 4. PP operations and discriminations for the type $A(T)$

4.4 Arrays of variable size

Assume that we are given a PP-type T. Instead of using a single word p to point to an array of fixed size we can use the two words designating array segments in an representation predicate for the PP-type $A(T)$ of variable size arrays:

$$a \sim_s^{A(T)} t \leftrightarrow \exists s_1 \exists s_2(s \doteq s_1 \star s_2 \wedge a \simeq_{s_1}^{A(T)} s_2 \mathrel{\widehat{\approx}} t)\ .$$

The basic high-level CL operations on $A(T)$ are $0\colon A(T)$, $Singl = \lambda x.[x]\colon T \to A(T)$, and $\oplus\colon A(T) \times A(T) \to A(T)$. Basic discriminations are the list splitting typed as $Dconc\colon A(T) \times W \to \bigcup\{\{0\}, A(T) \times A(T)\}$ and the taking of a singleton typed as $Dsingl\colon A(T) \to T$.

Empty list 0 is translated to the empty segment $(0,0)\colon W^2$. The clausal definitions of PP operations/discriminations corresponding to the remaining CL operations/discriminations are given in Fig. 4. The PP concatenation $\oplus'\colon W^2 \times W^2 \to W^2$ has the following property:

$$a \simeq_s^{A(T)} s_1 \mathrel{\widehat{\approx}} t \wedge b \simeq_s^{A(T)} s_2 \mathrel{\widehat{\approx}} u \wedge adj(t,u) \to \\ a \oplus b \simeq_s^{A(T)} s_1 \star s_2 \mathrel{\widehat{\approx}} t \oplus' u$$

where the predicate $adj(t,u)$ holds iff t, u are designators of array segments and they are *adjacent*, i.e., either one of them has length 0 or they are of a form $t = p,k$ and $u = p+k,n$. The condition on $\oplus'$ restricts during the translation the high-level operation $\oplus$ to lists whose PP array segment designators are adjacent. The PP operation $Singl'\colon W \times W^2 \times S \to S$ has the property:

$$s \doteq s_1 \star s_2 \wedge x \sim_{s_1}^{T} p \wedge s_2 \mathrel{\widehat{\approx}} t \wedge L(t) = 1 \to \\ \exists s_2'(Singl'(p,t,s) \doteq s_1 \star s_2' \wedge [x] \simeq_{s_1}^{A(T)} s_2' \mathrel{\widehat{\approx}} t)$$

and its condition restricts the high-level operation $[x]$ to such positions where the array segment designator of the result is $t = q,1$ with $s[q]$ existing and vacant (i.e., its value is not needed anymore and can be overwritten).

The two PP discriminators reversing the effects of $\oplus'$ and $Singl'$ are typed as $Dconc'\colon W^2 \times W \to \bigcup\{\{0\}, W^2 \times W^2\}$ and $Dsingl'\colon W^2 \times S \to W$.

$$Revv(0) = 0$$
$$Revv([x] \oplus 0) = [x]$$
$$Revv([x] \oplus a \oplus [y]) = [y] \oplus Revv(a) \oplus [x]$$

$$Revv'(t,s) = s \;\leftarrow Dconc'(t,1) = 1,0$$
$$\leftarrow \exists s_1 \exists s_2 \exists a(s \doteq s_1 \star s_2 \wedge a \simeq_{s_1}^{A(T)} s_2 \div t)$$
$$Revv'(t,s) = s \;\leftarrow Dconc'(t,1) = 2,(q,1),t_1 \wedge L(t_1) = k+1 \wedge$$
$$Dconc'(t_1,k) = 1,0$$
$$\leftarrow \exists s_1 \exists s_2 \exists a(s \doteq s_1 \star s_2 \wedge a \simeq_{s_1}^{A(T)} s_2 \div t)$$
$$Revv'(t,s) = s_2 \leftarrow Dconc'(t,1) = 2,(q,1),t_1 \wedge L(t_1) = k+1 \wedge$$
$$Dconc'(t_1,k) = 2,t_2,(p,1) \wedge$$
$$Revv'(t_2,s) = s_1 \wedge s_1[q{:=}s_1[p]][p{:=}s_1[q]] = s_2$$
$$\leftarrow \exists s_1 \exists s_2 \exists a(s \doteq s_1 \star s_2 \wedge a \simeq_{s_1}^{A(T)} s_2 \div t)$$

Figure 5. High-level and PP array reversal programs

4.5 In-place list reversal programmed properly

The high-level version of the list reversal function $Revv(a)$ is given in Fig. 5. Note that the program is only slightly more complicated than the definition of $Rev(a)$ needing $O(L(a)^2)$ pairing operations. There should be no doubt that this programs is significantly more readable than its naive version in Fig. 3. Moreover, the program is verified as $Revv(a) = Rev(a)$ by a trivial induction proof on the length of the list a.

The typing $Revv : (a : A(T) \to b : A(T))$ with $a \simeq b$ generates the $O(L(a))$ PP function $Revv' : W^2 \times S \to S$. Its (total) correctness property is automatically proved as:

$$a \simeq_s^{A(T)} s_1 \div t \to$$
$$\exists s_1'(Revv'(t, s \star s_1) = s \star s_1' \wedge Revv(a) \simeq_s^{A(T)} s_1' \div t) .$$

$$Part([p] \oplus 0) = 0,[p],0$$
$$Part([p] \oplus a \oplus [v]) = l,r,(g \oplus [v]) \qquad \leftarrow p \le v \wedge$$
$$Part([p] \oplus a) = l,r,g$$
$$Part([p] \oplus 0 \oplus [v]) = [v],[p],0 \qquad \leftarrow p > v$$
$$Part([p] \oplus [w] \oplus b \oplus [v]) = ([v] \oplus l),r,g \leftarrow p > v \wedge$$
$$Part([p] \oplus b \oplus [w]) = l,r,g$$

$$Qs(0) = 0$$
$$Qs(a) = Qs(l) \oplus r \oplus Qs(g) \leftarrow a \neq 0 \wedge Part(a) = l,r,g$$

Figure 6. High-level array Quicksort program

4.6 Quicksort

Fig. 6 gives a high-level CL definition of the well-known Quicksort program. The application $Qs(b)$ for $b = [p] \oplus a$ calls on the partition function *Part* for which one proves by induction on $L(a)$:

$$\exists l \exists g(Part([p] \oplus a) = l,[p],g \wedge \max(l) < p \wedge p \le \min(g) \wedge a \sim l \oplus g)$$

where the predicate $a \sim b$ holds if the list a is a permutation of the list b (see [7]). The Quicksort program is verified by proving $a \sim Qs(a) \wedge Ord\,Qs(a)$ also by induction on $L(a)$. Here the predicate $Ord(a)$ holds if the list a is non-decreasing.

The translation of Qs into an efficient PP program is triggered by the typing $Qs\colon (a\colon A(N) \to b\colon A(N))$ with $a \simeq b$ and $Part\colon (a\colon A(N) \to l\colon A(N) \times r\colon A(N) \times g\colon A(N))$ with $a \simeq l \oplus r \oplus g$. The last specifies that the three output array segments of $Part'$ should in that order overlay the single input array segment which has the pivot element p at the beginning. For instance, the clause for $Part'$ corresponding to the last clause of *Part* will maintain the adjacency by moving for the recursive call the pivot p to w's slot which will go to v's slot which will go to p's slot and the last slot will be omitted from the call.

The correctness of the translation is proved automatically:

$$s \doteq s_1 \star s_2 \wedge a \simeq_{s_1}^{A(N)} s_2 \div t \to$$
$$\exists s_2'(Qs'(t,s) \doteq s_1 \star s_2' \wedge Qs(a) \simeq_{s_1}^{A(N)} s_2' \div t) .$$

$$Mrgl(0,b) = b$$
$$Mrgl([u] \oplus a, 0) = [u] \oplus a$$
$$Mrgl([u] \oplus a, [v] \oplus b) = [u] \oplus Mrgl(a, [v] \oplus b) \leftarrow u \le v$$
$$Mrgl([u] \oplus a, [v] \oplus b) = [v] \oplus Mrgl([u] \oplus a, b) \leftarrow u > v$$

$$Msl(0) = 0$$
$$Msl(a \oplus b) = Mrgl(Msl(a), Msl(b)) \leftarrow 0 < L(a) \le L(b) \le L(a)+1$$

Figure 7. High-level list merge sort program

$$Mrg(0,b,c) = Copy(b,c) \qquad \leftarrow L(0 \oplus b) = L(c)$$
$$Mrg([u] \oplus a, 0, c) = Copy([u] \oplus a, c) \qquad \leftarrow L([u] \oplus a \oplus 0) = L(c)$$
$$Mrg([u] \oplus a, [v] \oplus b, [\_] \oplus c) = [u] \oplus Mrg(a, [v] \oplus b, c) \leftarrow u \le v$$
$$\leftarrow L([u] \oplus a \oplus [v] \oplus b) = L([\_] \oplus c)$$
$$Mrg([u] \oplus a, [v] \oplus b, [\_] \oplus c) = [v] \oplus Mrg([u] \oplus a, b, c) \leftarrow u > v$$
$$\leftarrow L([u] \oplus a \oplus [v] \oplus b) = L([\_] \oplus c)$$

$$Ms_1(0,c) = 0 \qquad \leftarrow L(0) = L(c)$$
$$Ms_1(a \oplus b, c \oplus d) = Mrg(Ms_2(a,c), Ms_2(b,d), a \oplus b)$$
$$\leftarrow 0 < L(a) \le L(b) \le L(a)+1 \wedge L(c) = L(a)$$
$$\leftarrow L(a \oplus b) = L(c \oplus d)$$

$$Ms_2(0,c) = c \qquad \leftarrow L(0) = L(c)$$
$$Ms_2(a \oplus b, c \oplus d) = Mrg(Ms_1(a,c), Ms_1(b,d), c \oplus d)$$
$$\leftarrow 0 < L(a) \le L(b) \le L(a)+1 \wedge L(c) = L(a)$$
$$\leftarrow L(a \oplus b) = L(c \oplus d)$$

$$Ms(a) = Ms_1(a, New\,L(a))$$

Figure 8. High-level array merge sort program

4.7 Merge sort

Fig. 7 contains a list version of the merge sort program. The function $Msl(c)$ repeatedly splits its input list c into two lists of roughly equal size by performing a discrimination for which exactly one of $L(c) = 0$ or $c = a \oplus b \wedge 0 < L(a) \leq L(b) \leq L(a)+1$ for unique a and b holds. We can prove (by hand) the correctness of *Msl* as $a \sim Msl(a) \wedge OrdMsl(a)$.

The call $Msl(c)$ will perform $O(\log L(c) \times L(c))$ pairing operations. This can be speeded up, and the argument c stored in an array, by providing in a second array of the same size auxiliary storage for the merges. Obviously, this cannot be done automatically and so we program a new version given in Fig. 8.

The functions of the new program seemingly do not do anything useful as they completely ignore their last arguments. Indeed, we can easily prove (by hand):

$$
\begin{gathered}
L(a) = L(b) \to Copy(a,b) = a \\
L(a)+L(b) = L(c) \to Mrgl(a,b) = Mrg(a,b,c) \\
L(a) = L(c) \to Msl(a) = Ms(a,c) = Ms_1(a,c) = Ms_2(a,c)\ .
\end{gathered}
$$

This trivially transfers the correctness of *Msl* to *Ms* verifying the latter as a sorting program,

The storage for the additional arguments will be overwritten with the results of the three functions. This is achieved by a translation triggered by the typing $Copy\colon (A(N) \times b\colon A(N) \to c\colon A(N))$ with $b \simeq c$, $Mrg\colon (A(N) \times A(N) \times c\colon A(N) \to d\colon A(N))$ with $c \simeq d$, $Ms_1\colon (a\colon A(N) \times A(N) \to d\colon A(N))$ with $a \simeq d$, $Ms_2\colon (A(N) \times c\colon A(N) \to d\colon A(N))$ with $c \simeq d$, and finally $Ms\colon (a\colon A(N) \to b\colon A(N))$ with $a \simeq b$.

We give here just the correctness property for the PP program $Ms_1'\colon N^2 \times N^2 \times S \to S$:

$$
\begin{aligned}
L(a) = L(c) \wedge a \simeq_s^{A(N)} s_1 \Bumpeq t \wedge c \simeq_s^{A(N)} s_2 \Bumpeq u \to \\
\exists s_1' \exists s_2' (Ms_1'(t,u,s \star s_1 \star s_2) = s \star s_1' \star s_2' \wedge \\
Ms_1(a,c) \simeq_s^{A(N)} s_1' \Bumpeq t \wedge s_2' \Bumpeq u)\ .
\end{aligned}
$$

Note that the store designated by u is modified in the recursive calls to Ms_2'.

$$
\begin{aligned}
& Bfn_1(n,0) = 0 \\
& Bfn_1(n,[\bullet]\oplus q) = [\bullet] \oplus Bfn_1(n,q) \\
& Bfn_1\left(n, \left[\frac{x}{l \mid r}\right] \oplus q\right) = \left[\frac{n}{l_1 \mid r_1}\right] \oplus q_1 \\
& \qquad \leftarrow Bfn_1(n+1, q \oplus [l] \oplus [r]) = q_1 \oplus [l_1] \oplus [r_1] \\
& \\
& Bfn(t) = Bfn_1(0,[t])
\end{aligned}
$$

Figure 9. Breadth-first numbering of binary trees

4.8 Breadth-first tree numbering

Here is an interesting problem analyzed by Okasaki [8]:

> Given a labeled binary tree, create a new tree of the same shape but with labels replaced by consecutive numbers in breadth-first order.

Okasaki's surprisingly difficult to discover solution given in Fig. 9 uses a queue storing a forest of trees. He does not present the solution in this way (knowing that it would be inefficient because of the required access to the ends of lists), and uses instead an abstract data type *queue*. He then proceeds with its implementation by a declarative double-ended queue.

$$
\begin{aligned}
& \overline{Bfn}_1(n,0,e) = 0 && \leftarrow C(0,e) \\
& \overline{Bfn}_1(n,[\bullet]\oplus q,e) = [\bullet] \oplus \overline{Bfn}_1(n,q,e) && \leftarrow C([\bullet]\oplus q,e) \\
& \overline{Bfn}_1\left(n, \left[\frac{x}{l \mid r}\right] \oplus q, [\_{}_1] \oplus [\_{}_2] \oplus e\right) = \left[\frac{n}{l_1 \mid r_1}\right] \oplus q_1 && \\
& \qquad \leftarrow \overline{Bfn}_1(n+1, q \oplus [l] \oplus [r], e) = q_1 \oplus [l_1] \oplus [r_1] && \\
& && \leftarrow C\left(\left[\frac{x}{l \mid r}\right] \oplus q, [\_{}_1] \oplus [\_{}_2] \oplus e\right) \\
& && \\
& \overline{Bfn}(t) = \overline{Bfn}_1(0,[t],e) \leftarrow New\sharp([t]) = [\_] \oplus e && \leftarrow Bt(t)
\end{aligned}
$$

Figure 10. Breadth-first numbering of binary trees without allocation

Having gone through the preceding examples, the reader immediately realizes that there is no need for such queues, all we need is to implement the queue in an array. So the first step is to remove the allocation of new queue slots by the two singleton operations in the argument of the recursive call in the last clause for Bfn_1. Note that the two singleton operations in the last two clauses for Bfn_1 used in the construction of the result can be done in place. The remaining four occurrences of singletons in Bfn_1 are pattern matched in discriminations by *Dsingl*.

The length of the needed queue can be computed. Preparatory to that we introduce into PA a predicate $Bts(a)$ holding iff a is a forest containing binary trees (with numbers as labels), i.e., if every element t of a satisfies the predicate $Bt(t)$ with a clausal definition:

$$
\begin{aligned}
& Bt(\bullet) \\
& Bt\left(\frac{n}{l \mid r}\right) \leftarrow Bt(l) \wedge Bt(r)\ .
\end{aligned}
$$

We then introduce into PA the *tree size* function $\sharp(a)$ which for a forest of binary trees a counts the total number of occurrences of constructors $\bullet$ and $\frac{\cdot}{\cdot \mid \cdot}$ in all elements of a. The number of singleton slots allocated in the call $Bfn(n,q)$ is then k where $L(q)+k = \sharp(q)$.

Similarly as with the merge sort example, we now reprogram (by hand) the numbering program adding to Bfn_1 a new dummy argument e whose slots will be used in the extension of q. The program is given in Fig. 10. The condition C on $\overline{Bfn}_1$ is defined as follows:

$$
C(q,e) \leftrightarrow Bts(q) \wedge L(q)+L(e) = \sharp(q)\ .
$$

The condition is established in the initial call of $\overline{Bfn}_1$ from $\overline{Bfn}(t)$ from its condition $Bt(t)$ and it is maintained in the recursive calls to $\overline{Bfn}_1$. We then correlate the old functions with new ones by easily proving (by hand):

$$
\begin{gathered}
C(q,e) \to Bfn_1(n,q) = \overline{Bfn}_1(n,q,e) \\
Bt(t) \to Bfn(t) = \overline{Bfn}(t)\ .
\end{gathered}
$$

We trigger the translation to PP with the typing

$$
\overline{Bfn}_1\colon (N \times q\colon A(N) \times e\colon A(n) \to r\colon A(N)) \text{ with } r \oplus e \simeq q \oplus e
$$

and $\overline{Bfn}\colon N \to N$. The storage sharing instruction $r \oplus e \simeq q \oplus e$ tells the translator that the array segments corresponding to lists q and e

will be adjacent, and that the result r should overlay q. Note that we did not give any special PP representation to binary trees, they are represented in the standard way through the PP-type N. The partial correctness property for the function $\overline{Bfn}'$ can be generated and proved automatically:

$$
\begin{aligned}
&Bt(t) \land t \sim_{s_1}^{N} p \land s_2 \Bumpeq h \land \overline{Bfn}' 0(p, h, s_1 \star s_2) = q, h', s' \to \\
&\qquad \exists s'_1 \exists s'_2 (s' \doteq s_1 \star s'_1 \star s'_2 \land Bfn(t) \sim_{s_1 \star s'_1}^{N} q \land s'_2 \Bumpeq h') \ .
\end{aligned}
$$

Note that the store s'_1 is used for the representation of $Bfn(t)$, and that the argument t is not overwritten.

4.9 Representation of binary trees

The reader now surely asks whether it would be possible to do the breadth-first numbering of binary trees from the preceding paragraph by also overwriting the binary trees in place. Probably the simplest way to go about it is to design for a given PP-type T a PP-type $Bt(T)$ implementing binary trees with labels from T. Its representation predicate is defined as follows:

$$
\begin{aligned}
&\bullet \sim_s^{Bt(T)} 0 \quad \leftarrow 0, s \colon W \times S \\
&\frac{n}{l \mid r} \sim_s^{Bt(T)} p \leftarrow \frac{n}{l \mid r}, s \colon W \times S \land \\
&\qquad\qquad s \doteq [p \mapsto q_1; q_2; q_3] \star s_1 \star s_2 \star s_3 \land \\
&\qquad\qquad n \sim_{s_1}^{T} q_1 \land l \sim_{s_2}^{Bt(T)} q_2 \land r \sim_{s_3}^{Bt(T)} q_3 \ .
\end{aligned}
$$

We now select and type the basic operations on $Bt(T)$, for instance, the constructors $\bullet \colon Bt(T)$ and $\lambda nlr. \frac{n}{l \mid r} \colon T \times Bt(T) \times Bt(T) \to Bt(T)$ and similarly for a suitable discrimination on binary trees. We then design the corresponding PP operations and a discrimination, and prove properties connecting the two sets.

For simplicity we will work with binary trees with labels from W. After all, we do not need more labels for the numbering of binary trees. We abbreviate by B the PP-type $A(Bt(W))$ which will represent the forests of binary trees. We should also change the conditions on the functions in Fig. 10. The new condition C is

$$C(n, q, e) \leftrightarrow Bts(q) \land L(q) + L(e) = \sharp(q) \land n + \sharp(q) \div 2 \colon W$$

and the condition on $\overline{Bfn}$ is $Bt(t) \land \sharp([t]) \div 2 \colon W$. The new conditions are needed in order to guarantee that the $n+1$ operation in the body of $\overline{Bfn}_1$ does not overflow (we will need as many labels as there are inner nodes in the trees). The reader should note how the restriction of labels to W complicates the programming, there are new proof obligations to be satisfied.

The translation to PP is initiated by the typing $\overline{Bfn}_1 \colon (W \times q \colon B \times e \colon B \to r \colon B)$ with $r \oplus e \simeq q \oplus e$, but, we also need somehow to instruct the translator that the overwriting is 'deep', i.e., that also the trees in the forests should be overwritten in place.

With the deep sharing, the total correctness property should be:

$$
\begin{aligned}
&C(n, q, e) \land q \sim_{s_1}^{B} u \land e \sim_{s_2}^{B} v \land s \doteq s_1 \star s_2 \to \\
&\qquad \exists s'_1 \exists s'_2 \exists e' (\overline{Bfn}'_1(n, u, v, s) = s'_1 \star s'_2 \land \\
&\qquad\qquad \overline{Bfn}_1(n, q, e) \sim_{s'_1}^{B} u \land e' \sim_{s'_2}^{B} v) \ .
\end{aligned}
$$

Note that the representation predicates with $\sim$ are used instead of those with $\simeq$.

The typing $\overline{Bfn} \colon (t \colon Bt(W) \to r \colon Bt(W))$ with $t \simeq r$ initiates the translation with the partial correctness property:

$$
\begin{aligned}
&Bt(t) \land t \sim_{s_1}^{Bt(W)} u \land s_2 \Bumpeq h \land \overline{Bfn}'(u, h, s_1 \star s_2) = 1, h', s' \to \\
&\qquad \exists s' (s' \doteq s'_1 \star s'_2 \land \overline{Bfn}(t) \sim_{s'_1}^{Bt(W)} u \land s'_2 \Bumpeq h') \ .
\end{aligned}
$$

5 Conclusions

We have outlined a methodology for the automatic translation of high-level CL programs to low-level PP programs. PP programs are so close to imperative programs that their transformation to such is straightforward. The methodology is extensible because it allows for the definition of new PP-types efficiently representing various high-level objects. The first such type which should be designed is the record (structure) type. We have also outlined a methodology for the verification of high-level functions operating on lists in such a way that their PP translations will operate on arrays with the reuse of storage.

We have strong reasons to believe that it will be possible for the translator to formulate the correctness property for the translation and have it automatically proved. The first author's thesis will be concerned with the design of algorithms for the translation with the reusal of storage and with proving their correctness on a meta-theoretical level. The hardest part of the thesis will be its proof-theoretical part which will call for the design of an algorithm taking proofs in PA of properties of high-level functions and yielding the proof of the correctness properties of their translations. The correctness of the algorithm, must be proved again at the meta-theoretical level. We think that this will be probably the first application of proof theory to a real practical problem.

Note that the meta-theoretical proof is not directly connected to the proof carrying issue. The generated proof of the correctness property together with the proofs of high level properties can be mechanically checked and, if correct, their correctness will be independent of possible mistakes in the meta-theoretical proofs. Of course, if the proof transfer algorithm is wrong, or if there is a bug in the intelligent proof assistant of CL, then the proofs may fail the check, but at least we will know that something is wrong.

6 References

[1] A. Bloss, Path analysis and the optimization of nonstrict functional languages. *ACM Transactions on Programming Languages and Systems*, 16(3):328–369, 1994.

[2] *The programming language and proof system CL*, www.ii.fmph.uniba.sk/cl/ .

[3] *The programming language and proof system CL*, www.ii.fmph.uniba.sk/cl/online/ .

[4] P. O'Hearn, J. Reynolds, H. Yang, Local Reasoning about Programs that Alter Data Structures. In *Proceedings of CSL'01, Paris, 2001*, LNCS 2142, pages 1–19, Springer-Verlag, 2001.

[5] N. Shankar, *Efficiently Executing PVS*. SRI International, Menlo Park, CA, 1999.

[6] J. Kľuka, *A Simple Semantics for Destructive Updates*. ESSLLI'03, Vienna, 2003. To appear.

[7] J. Komara, *Specification and Verification of Programs*, 2001, www.ii.fmph.uniba.sk/cl/courses/ .

[8] C. Okasaki, Breadth-First Numbering: Lessons from a Small Exercise in Algorithm Design. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ACM Press, 2000, pages 131–136.

[9] A.V. Sastry, W. Clinger, Z. M. Ariola. Order-of-evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*. University of Copenhagen, Denmark, June 1993.

[10] J. Shoenfield, *Mathematical Logic*. Addison-Wesley, 1967.

[11] P. J. Voda, *What Can we Gain by Integrating a Language Processor with a Theorem Prover?*, www.fmph.uniba.sk/∼voda, 2003.

[12] P. J. Voda, *Grundlagenstreit in the Theory of Programming*, www.fmph.uniba.sk/∼voda, 2003.