Type-directed List Compaction

Nobuaki Yoshida
Research Institute for Mathematical Sciences
Kyoto University
nyoshi@kurims.kyoto-u.ac.jp

Abstract

This paper presents a *type-directed list compaction* method, which statically realizes vector-style list representation without increasing runtime overheads. To achieve this, we define a compilation target language where vector-style lists are constructed by filling preallocated memory locations imperatively. We develop a type system for this language, which allows us to exploit such imperative feature without destroying functional semantics. We then develop a compilation algorithm from a source language with ordinal list representation to the target language. Our preliminary experiment shows approximately 5-13% improvement in execution time by our method even with bytecode interpretation.

1 Introduction

Automatically allocated dynamic data structures are one of important features of functional programming languages. With the feature of higher-order functions, they enable flexible and high-level manipulation of recursively defined data structures. However, they are less efficient than imperative data structures, such as vectors, in both memory usage and execution time. In a conventional implementation, dynamic data structures are represented as pointers maintained automatically by a runtime system. While this makes the language flexible, extra memory spaces for pointers are required and memory reference is unavoidable to access data elements. This is in contrast with imperative data structures, which are less flexible but allow compact representation and efficient manipulation through pointer arithmetic.

In spite of inefficiency, dynamic data structures are essential to functional programming languages, and their optimization is an important implementation issue. Among them, list structure is most frequently used, and its optimization has been an important research topic. In 1970's, "cdr-coding" [8, 2, 3, 4, 12] was proposed for Lisp to eliminate some cdr pointers. In this representation, a cdr part is placed directly in the next memory location of its car part. To distinguish cdr-coded cons cells from usual ones, extra tag bits exist in each cell. Original cdr-coding method was further refined in [1, 12] to support runtime construction of cdr-coded lists. Cdr-coding has succeeded in optimization of memory usage, but not in reducing execution time: cdr-coding requires expensive tag checking when lists are constructed and traversed.

Although "cdr-coding" style of list representation is not commonly adopted in current statically typed languages, its general idea is still potentially attractive. If we can eliminate the runtime overhead mentioned above, this style of representation will achieve efficient list manipulation in time and space. In this paper, we present one such method, *type-directed list compaction*, for a language like ML [13, 11]. In our compact representation, a cdr part is placed directly in next memory location of its car cell, and can be accessed sequentially with simple pointer arithmetic. This representation can be combined with a conventional garbage collection

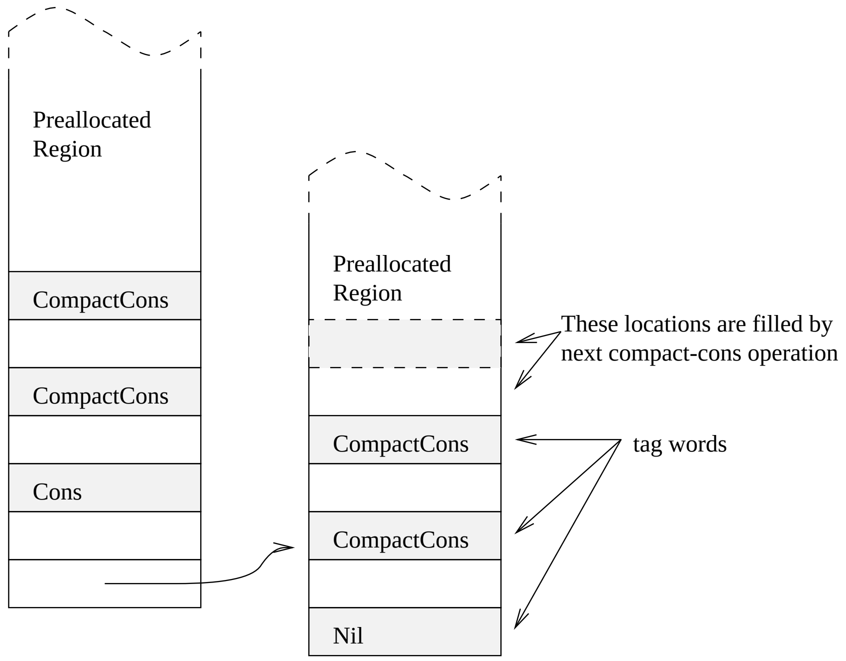

Figure 1: Compact list representation

algorithm. We develop this method as follows. We first define an implementation language, Λ^{CL}. In this language, constructors of nil and (ordinal) cons preallocate a consecutive memory block which is filled with car values by an imperative operator, *compact-cons*. Figure 1 illustrates runtime memory arrangement of this representation. To obtain semantically correct execution with such an unsafe operator, we give a static type system for Λ^{CL} with a mechanism similar to that of linear logic [6]. We then develop a type inference and compilation algorithm from a language with conventional list constructors to Λ^{CL}. With this algorithm, a program can be compiled to safe and efficient code using compact list representation. For example, consider the following `map` function(written in ML-like syntax):

```
fun map(f,[]) = []
|   map(f,h::t) = (f h)::(map(f, t))
```

This program is compiled to the code that works as follows. For the case of `Nil`, it preallocates a memory block at the same time of constructing nil cell. For the case of `Cons`, it calls `map` recursively, and constructs a compact cons cell by filling the next location of the returned address with the result of `(f h)`.

In this program, cons operator occurs only once, and therefore, its translation is straightforward. However, in general, obtaining a correct translation is a subtle problem. To understand the difficulty, consider the following example.

```
fun f 0 = []
|   f n = let
              val x = f (n-1)
          in
              (#2 (n::x, (n+1)::x))
          end
```

If we translate both occurrences of `cons` to compact-cons, one of `n` or `n+1` will be overwritten by the other value incorrectly. We solve this problem by defining a type system which restricts incorrect use of compact-cons as above, and developing an algorithm to reconstruct a correct typing for a source program in this type system. A safe and efficient code is, then, extracted from the typing obtained by the type reconstruction algorithm.

This paper is organized as follows. In Section 2, we present a source language, Λ^L, and an implementation language, Λ^{CL}. We also present their type systems. In Section 3, we give an operational semantics for this language and prove the type soundness. In Section 4, we also present an algorithm which translates Λ^L term to Λ^{CL} term, and prove that it preserves typing. Section 5 describes implementations. In particular, with the existence of compact-cons cells, garbage collection is a non trivial issue. We will give a solution to this problem. This section also shows results of a preliminary test of our method.

1.1 Related Work

In the field of statically typed programming languages, some optimizations of list representation are proposed. The most relevant works to ours are those on list unrolling [7, 15]. In their approach, each cons cell can contain up to n elements, and a list is a chain of such cells. They eliminates runtime overheads by introducing different cell type for each size of cons cells, and statically inserting case branches on types of cells. However, this representation increases the number of types as n becomes bigger. This increases overheads in case branches, and is subject to code explosion. In our approach, each cons cell may have arbitrary length, but has the same type. Safe and efficient execution of programs is achieved by analyzing the behavior of cons operator. As a consequence, our approach does not incur these problems.

There have been a number of proposals for efficient memory management based on linear type systems. Among them, followings have some similarity to ours. A quasi-linear value [9] is a linear value which can be accessed locally many times. This type system provides access to a linear value in a more relaxed manner than the original linear type system on which it is based. This is achieved by taking evaluation order into account. In [14], a data structure with a hole is introduced into a call-by-value functional language. With this data structure, functions such as `map` and `append` can be compiled to tail recursive functions which destructively update a hole. They restrict destructive update of a hole at most once using a linear type system. In [16, 9], the notion of "use" is introduced to control how often a value can be accessed. We use similar attribute to guarantee memory locations to be written at most once. However, approaches taken in these works are not directly applicable to the problem considered in this paper. We need to represent an availability of unused memory location which cannot be expressed as an attribute of a type. We solve this problem by giving an attribute for global behavior of data constructors.

2 Source and Target Languages

2.1 Source Language: Λ^L

To present our method, we consider the following source language with lists, Λ^L. Its types and terms are given as follows:

$$\begin{array}{rcl} \tau & ::= & b \mid \tau \to \tau \mid \tau\ \texttt{list} \\ M & ::= & c^b \mid x \mid \texttt{fix}\ f\ x.M \mid MM \end{array}$$

$$\text{Const}\quad \Gamma \rhd c^b : b$$

$$\text{Var}\quad \Gamma\{x : \tau\} \rhd x : \tau$$

$$\text{Fix}\quad \frac{\Gamma\{f : \tau \to \tau', x : \tau\} \rhd M : \tau'}{\Gamma \rhd \texttt{fix}\ f\ x.M : \tau \to \tau'}$$

$$\text{App}\quad \frac{\Gamma_1 \rhd M_1 : \tau \to \tau' \qquad \Gamma_2 \rhd M_2 : \tau}{\Gamma_1 \cup \Gamma_2 \rhd M_1 M_2 : \tau'}$$

$$\text{Nil}\quad \Gamma \rhd \texttt{Nil} : \tau\ \texttt{list}$$

$$\text{Cons}\quad \frac{\Gamma_1 \rhd M_1 : \tau \qquad \Gamma_2 \rhd M_2 : \tau\ \texttt{list}}{\Gamma_1 \cup \Gamma_2 \rhd \texttt{Cons}(M_1,M_2) : \tau\ \texttt{list}}$$

$$\text{Case}\quad \frac{\Gamma_1 \rhd M_1 : \tau\ \texttt{list} \qquad \Gamma_2 \rhd M_2 : \tau' \qquad \Gamma_2\{x : \tau, y : \tau\ \texttt{list}\} \rhd M_3 : \tau'}{\Gamma_1 \cup \Gamma_2 \rhd \texttt{Case}\ M_1\ \texttt{of Nil} \Rightarrow M_2,\ \texttt{Cons}(x,y) \Rightarrow M_3 : \tau'}$$

Figure 2: Type System for Λ^L

$$\begin{array}{ll} | & \texttt{Nil} \mid \texttt{Cons}(M,M) \\ | & \texttt{Case}\ M\ \texttt{of Nil} \Rightarrow M,\ \texttt{Cons}(x,y) \Rightarrow M \end{array}$$

where, $\texttt{fix}\ f\ x.M$ is a function $\lambda x.M$ with f bound to this function itself. Typing rules are shown in Figure 2.

2.2 Implementation Language: Λ^{CL}

The set of terms of Λ^{CL} is given as follows:

$$\begin{array}{rcl} M & ::= & c^b \mid x \mid \texttt{fix}\ f\ x.M \mid MM \\ & | & \texttt{Nil} \mid \texttt{Cons}_0(M,M) \mid \texttt{Cons}_1(M,M) \\ & | & \texttt{Case}\ M\ \texttt{of Nil} \Rightarrow M,\ \texttt{Cons}_0(x,y) \parallel \texttt{Cons}_1(x,y) \Rightarrow M \end{array}$$

$\texttt{Nil}$ and $\texttt{Cons}_0(M_1,M_2)$ correspond to nil and usual cons cell respectively. $\texttt{Cons}_1(M_1,M_2)$ is a compact-cons operation, which writes a value of M_1 in the next memory location of M_2 without checking whether some value exists in that location or not. A case expression binds car and cdr part to x and y in M_3, respectively. To emphasize that there are two access methods corresponding to different list constructors, $\texttt{Cons}_0$ and $\texttt{Cons}_1$, we adopt this syntax.

To guarantee semantically correct execution of a program written in this language, we must restrict application of compact-cons operation to a *next unused location* of a list block. In the rest of this section, we develop a static type system which realizes such restriction with a mechanism similar to that of linear logic.

First, we define a set of types for Λ^{CL} as follows.

$$\begin{array}{rcl} \tau & ::= & b \mid \sigma \to \sigma \mid \sigma\ \texttt{list} \\ \sigma & ::= & \tau^u \\ u & ::= & 1 \mid 0 \end{array}$$

In our type system, a type is accompanied with a *use attribute*, u, which is similar to a "use" in [16, 9]. If a list can become a cdr part of a compact-cons cell, that is, its has a next unused location, it has an attribute *1*. Otherwise, it has an attribute *0*. On use attributes, we define additions and an inequality relation as follows.

$$1 + 0 = 0 + 1 = 1$$

$$0 + 0 = 0$$

$$1 > 0$$

Note that $1 + 1$ is undefined.

The type system for Λ^{CL} is defined as a proof system to derive a judgment of the form $\Gamma \rhd M : \sigma$. Γ assigns σ to a variable. The set of typing rules is given in Figure 3. In rule Cons, the attribute *0* of M_2 means M_2 can be used as a cdr part of an ordinal cons cell. On the other hand, in rule CompactCons, to execute compact-cons operation safely with $\texttt{Cons}_1$, M_2 must have a next unused location, which is guaranteed by the attribute *1*. In rule App, the attribute of argument type of the function and that of the actual argument must be equal. The rule Var allows that a value with next unused location can be used as a ordinal value. In rules App, Cons, CompactCons, and Case, which combine two or more sub terms, variables can be used as a value having next unused location at most once in one of sub terms. To guarantee this, in these rules, type assignments of sub terms are merged by the following operator, $\oplus$.

$$\Gamma_1 \oplus \Gamma_2 \overset{def}{\Longleftrightarrow}$$

$$\forall x \in Dom(\Gamma_1) \cup Dom(\Gamma_2),\ \Gamma_1 \oplus \Gamma_2(x) = \begin{cases} \tau^{u_1+u_2} & (x \in Dom(\Gamma_1) \cap Dom(\Gamma_2), \Gamma_i(x) = \tau^{u_i}) \\ \Gamma_1(x) & (x \in Dom(\Gamma_1), x \notin Dom(\Gamma_2)) \\ \Gamma_2(x) & (x \in Dom(\Gamma_2), x \notin Dom(\Gamma_1)) \end{cases}.$$

The type of elements in rules Nil, Cons, and CompactCons, and the types of free variables, except for f and x, of M', in rule Fix are restricted to τ^0. This is to prevent next unused locations from being copied through data structures and closures.

3 Operational Semantics and Type Soundness

In this section, we present an operational semantics for the language in the style of natural semantics, and prove soundness of our type system with respect to the operational semantics.

The operational semantics is given as a set of rules to derive the following evaluation relation:

$$E, H \models M \Downarrow v, H'$$

This relation indicates that, under environment E and store environment H, M is evaluated to a value v yielding modified store environment H'.

The set of values is defined as follows.

$$v \quad ::= \quad c^b \mid Clos(f,x,M,E) \mid \texttt{list}(i,j) \mid wrong$$

$Clos(f,x,M,E)$ is a closure where f is a variable bound to itself in M. $\texttt{list}(i,j)$ is a value for lists, and is a pointer to a *store environment*, H, for lists. *wrong* represents runtime error.

An environment E is a finite map from variables to values. We write $\{x_1 \leftarrow v_1, \cdots, x_i \leftarrow v_i, \cdots, x_n \leftarrow v_n\}$ for the environment that maps x_i to v_i, and write $E\{x \leftarrow v\}$ for the environment E' such that $E'(x) = v$ and $E'(y) = E(y)$ for $\forall y \in Dom(E), y \neq x$.

$$\text{Const} \quad \Gamma \rhd c^b : b^0$$

$$\text{Var} \quad \Gamma\{x : \tau^u\} \rhd x : \tau^{u'} \qquad u \geq u'$$

$$\text{Fix} \quad \frac{\Gamma\{f : (\tau^u \to \tau'^{u'})^0, x : \tau^u\} \rhd M : \tau'^{u'} \qquad \forall y \in Dom(\Gamma), \Gamma(y) = \tau^0}{\Gamma \rhd \mathtt{fix}\ f\ x.M : (\tau^u \to \tau'^{u'})^0}$$

$$\text{App} \quad \frac{\Gamma_1 \rhd M_1 : (\tau^u \to \tau'^{u'})^0 \qquad \Gamma_2 \rhd M_2 : \tau^u}{\Gamma_1 \oplus \Gamma_2 \rhd M_1 M_2 : \tau'^{u'}}$$

$$\text{Nil} \quad \Gamma \rhd \mathtt{Nil} : (\tau^0\ \mathtt{list})^u$$

$$\text{Cons} \quad \frac{\Gamma_1 \rhd M_1 : \tau^0 \qquad \Gamma_2 \rhd M_2 : (\tau^0\ \mathtt{list})^0}{\Gamma_1 \oplus \Gamma_2 \rhd \mathtt{Cons}_0(M_1,M_2) : (\tau^0\ \mathtt{list})^u}$$

$$\text{CompactCons} \quad \frac{\Gamma_1 \rhd M_1 : \tau^0 \qquad \Gamma_2 \rhd M_2 : (\tau^0\ \mathtt{list})^1}{\Gamma_1 \oplus \Gamma_2 \rhd \mathtt{Cons}_1(M_1,M_2) : (\tau^0\ \mathtt{list})^u}$$

$$\text{Case} \quad \frac{\Gamma_1 \rhd M_1 : (\tau^0\ \mathtt{list})^0 \qquad \Gamma_2 \rhd M_2 : \tau'^{u'} \qquad \Gamma_2\{x : \tau^0, y : \tau^0\ \mathtt{list}^0\} \rhd M_3 : \tau'^{u'}}{\Gamma_1 \oplus \Gamma_2 \rhd \mathtt{Case}\ M_1\ \mathtt{of}\ \mathtt{Nil} \Rightarrow M_2,\ \mathtt{Cons}_0(x,y) \parallel \mathtt{Cons}_1(x,y) \Rightarrow M_3 : \tau'^{u'}}$$

Figure 3: Type system for Λ^{CL}

H models physical memory space for list cells (ranged over by hv), and is an array of *store block* (ranged over by hb) which forms a consecutive list cell in compact representation. hv, hb and H are defined as follows.

$$\begin{array}{rcl} hv & ::= & \mathtt{nil} \mid \mathtt{ncons}(v,i,j) \mid \mathtt{fcons}(v) \\ hb & ::= & [hv_0, \cdots, hv_j, \cdots, hv_k] \\ H & ::= & [0 : hb, \cdots, i : hb, \cdots, n : hb] \end{array}$$

`nil` is a nil cell. $\mathtt{ncons}(v,i,j)$ is a usual `cons` cell where v is the `car` part, and (i,j) is the pointer to its `cdr` part. $\mathtt{fcons}(v)$ is a compact cons cell where v is the `car` part. Its `cdr` part directly follows this cell, and therefore,the cdr part pointer is unnecessary. $H(i,j)$ denotes jth element of ith store block in H. We assume sizes of H and each store block are sufficiently large, that is, indices of H and store blocks used in the following development are less than n and k respectively. A value $\mathtt{list}(i,j)$ is a pointer to $H(i,j)$.

Figure 4 gives the definition of our operational semantics. In addition to these rules, following rules which yield *wrong* are implicitly given: if the evaluation of one of its components specified in the rule yields *wrong* or does not satisfy its condition, or any of the operation used in the rule yields *wrong*, then, entire term yields *wrong*. In this definition, a notation $H + \{i : hb\}$ is a store environment obtained by replacing the ith block of H with hb. In the cases Nil and NCons, new block is allocated, and its first next unused location is filled with a value. (i,j) is a pointer to its cdr part. In the case FCons, instead of allocating new block, a value $\mathtt{fcons}(v)$ is written on $H(i, j+1)$, whether that location is already used or not.

Our operational semantics is unsafe. A compact-cons operation can incorrectly overwrite other value in memory space. However, for a well-typed term, we can show that its evaluation is always type consistent, and keeps memory space consistent. In the rest of this section, we prove this property, by showing the type soundness.

$$\text{Const} \quad E,H \models c^b \Downarrow c^b, H$$

$$\text{Var} \quad E,H \models x \Downarrow E(x), H$$

$$\text{Abs} \quad E,H \models \mathtt{fix}\ f\ x.M \Downarrow \mathit{Clos}(f,x,M,E), H$$

$$\text{App} \quad \frac{\begin{array}{c} E,H_1 \models M_1 \Downarrow \mathit{Clos}(f,x,M_1',E'),H_2 \qquad E,H_2 \models M_2 \Downarrow v,H_3 \\ E'\{f \leftarrow \mathit{Clos}(f,x,M_1',E'), x \leftarrow v\}, H_3 \models M_1' \Downarrow v', H_4 \end{array}}{E,H_1 \models M_1 M_2 \Downarrow v', H_4}$$

$$\text{Nil} \quad E,H \models \mathtt{Nil} \Downarrow \mathtt{list}(i,0), H + \{i : [\mathtt{Nil}]\} \quad (i : \text{an unused location in } H)$$

$$\text{NCons} \quad \frac{E,H_1 \models M_1 \Downarrow v,H_2 \qquad E,H_2 \models M_2 \Downarrow \mathtt{list}(i,j),H_3}{E,H_1 \models \mathtt{Cons}_0(M_1,M_2) \Downarrow \mathtt{list}(i',0), H_3 + \{i' : [\mathtt{ncons}(v,i,j)]\}}$$
$$(i' : \text{an unused location in } H)$$

$$\text{FCons} \quad \frac{E,H_1 \models M_1 \Downarrow v,H_2 \qquad E,H_2 \models M_2 \Downarrow \mathtt{list}(i,j),H_3 + \{i : [hv_0,\cdots,hv_j,hv_{j+1},\cdots]\}}{E,H_1 \models \mathtt{Cons}_1(M_1,M_2) \Downarrow \mathtt{list}(i,j+1), H_3 + \{i : [hv_0,\cdots,hv_j,\mathtt{fcons}(v),\cdots]\}}$$

$$\text{CaseNil} \quad \frac{\begin{array}{c} E,H_1 \models M_1 \Downarrow \mathtt{list}(i,j), H_2 + \{i : [hv_0,\cdots,hv_{j-1},\mathtt{Nil},\cdots]\} \\ E, H_2 + \{i : [hv_0,\cdots,hv_{j-1},\mathtt{Nil},\cdots]\} \models M_2 \Downarrow v, H_3 \end{array}}{E,H_1 \models \mathtt{Case}\ M_1\ \mathtt{of}\ \mathtt{Nil} \Rightarrow M_2,\ \mathtt{Cons}_0(x,y) \parallel \mathtt{Cons}_1(x,y) \Rightarrow M_3 \Downarrow v, H_3}$$

$$\text{CaseFCons} \quad \frac{\begin{array}{c} E,H_1 \models M_1 \Downarrow \mathtt{list}(i,j), H_2 + \{i : [hv_0,\cdots,hv_{j-1},\mathtt{fcons}(v),\cdots]\} \\ E\{x \leftarrow v, y \leftarrow \mathtt{list}(i,j-1)\}, H_2 + \{i : [hv_0,\cdots,hv_{j-1},\mathtt{fcons}(v),\cdots]\} \models M_3 \Downarrow v, H_3 \end{array}}{E,H_1 \models \mathtt{Case}\ M_1\ \mathtt{of}\ \mathtt{Nil} \Rightarrow M_2,\ \mathtt{Cons}_0(x,y) \parallel \mathtt{Cons}_1(x,y) \Rightarrow M_3 \Downarrow v, H_3}$$

$$\text{CaseNCons} \quad \frac{\begin{array}{c} E,H_1 \models M_1 \Downarrow \mathtt{list}(i,j), H_2 + \{i : [hv_0,\cdots,hv_{j-1},\mathtt{ncons}(v,i',j'),\cdots]\} \\ E\{x \leftarrow v, y \leftarrow \mathtt{list}(i',j')\}, H_2 + \{i : [hv_0,\cdots,hv_{j-1},\mathtt{ncons}(v,i',j'),\cdots]\} \models M_3 \Downarrow v, H_3 \end{array}}{E,H_1 \models \mathtt{Case}\ M_1\ \mathtt{of}\ \mathtt{Nil} \Rightarrow M_2,\ \mathtt{Cons}_0(x,y) \parallel \mathtt{Cons}_1(x,y) \Rightarrow M_3 \Downarrow v, H_3}$$

Figure 4: Operational semantics for Λ^{CL}

To prove the type soundness, we must first define value typing. In our system, a list value is expressed as a pointer to an element in H, and therefore, value typing depends on typing of values in H. To take account of this dependency, we follow [10] and introduce a store typing, $\mathcal{H}$, for H. $\mathcal{H}$ is defined as follows:

$$\mathcal{H} \quad ::= \quad \{i_0 : (\tau_0, j_0), \cdots, i_l : (\tau_l, j_l), \cdots, i_n : (\tau_n, j_n)\}$$

This $\mathcal{H}$ associates an index i_l of a store block in H with a pair of type τ_l of elements of list beginning at the block i_l, and the last index j_l of the block. $\mathcal{H}(i_l)$ denotes a type associated with index i_l in $\mathcal{H}$, that is, (τ_l, j_l).

Using this $\mathcal{H}$, value typing is given in the following form:

$$\mathcal{H} \models v : \sigma$$

denoting the fact that v has a type σ under $\mathcal{H}$. This relation is defined in Figure 5. We extend this relation to E as follows.

$$\mathcal{H} \models E : \Gamma \quad \stackrel{def}{\Longleftrightarrow} \quad \forall x \in Dom(\Gamma), \mathcal{H} \models E(x) : \Gamma(x)$$

The relation $H \models \mathcal{H}$ defined below implies that each element in a block has a type described in $\mathcal{H}$, and if another block is linked through `ncons` cell, then the linked block also has the same type.

$\mathcal{H} \models c^b : b^0$

$\mathcal{H} \models Clos(f,x,M,E) : (\tau^u \to \tau'^{u'})^0$
if $\Gamma \triangleright \texttt{fix}\ f\ x.M : (\tau^u \to \tau'^{u'})^0$ can be derived, and $\mathcal{H} \models E : \Gamma$

$\mathcal{H} \models \texttt{list}(i,j) : (\tau^0\ \texttt{list})^0$ if $\mathcal{H}(i) = (\tau^0, j')$ and $j \leq j'$

$\mathcal{H} \models \texttt{list}(i,j) : (\tau^0\ \texttt{list})^1$ if $\mathcal{H}(i) = (\tau^0, j)$

Figure 5: Value typing

$$H \models \mathcal{H} \quad \overset{def}{\iff} \quad \begin{array}{l} \forall i \in Dom(\mathcal{H}),\ \text{let } (\tau, j_i) = \mathcal{H}(i). \\ H(i,j) = \begin{cases} \texttt{Nil}\ \mathit{or}\ \texttt{ncons}(v,i',j') & (\text{If } j = 0) \\ \texttt{fcons}(v) & (\text{If } 0 < j \leq j_i) \end{cases} , \mathcal{H} \models v : \tau^0, \text{ and} \\ \mathcal{H}(i') = (\tau, j''). \end{array}$$

In order to formally define soundness property, we need another technical definition of a relation concerning $\mathcal{H}$. $\mathcal{H}'$ is an extension of $\mathcal{H}$ on E and Γ, written as $\mathcal{H}' \geq_{E,\Gamma} \mathcal{H}$, if they satisfy the following:

1. $\mathcal{H} \models E : \Gamma$
2. $Dom(\mathcal{H}') \supseteq Dom(\mathcal{H})$
3. $\forall i \in Dom(\mathcal{H})$, if $(\tau, j) = \mathcal{H}(i)$ and $(\tau, j') = \mathcal{H}'(i)$ then (1) $j' \geq j$, and (2) if $j' > j$ and $\exists x$ s.t. $E(x) = \texttt{list}(i,j)$, then, for at least one of such x, $\Gamma(x) = \tau^1$.

$\mathcal{H}' \geq_{E,\Gamma} \mathcal{H}$ means that, if a block is extended in $\mathcal{H}'$ and a pointer to the end of the block exists in E, then it has an attribute *1* in Γ.

We can now show the following soundness theorem for our calculus.

Theorem 1 *If $\Gamma \triangleright M : \tau^u$, $H \models \mathcal{H}$, $\mathcal{H} \models E : \Gamma$, and $E, H \models M \Downarrow v, H'$, then, $\exists \mathcal{H}'$ such that $\mathcal{H}' \geq_{E,\Gamma} \mathcal{H}$, and $H' \models \mathcal{H}'$ and $\mathcal{H}' \models v : \tau^u$ hold. If $u = 1$ and $\exists x$ such that $E(x) = v$, then at least one of such x has a type τ^1 in Γ.*

In addition to type preservation property, this theorem states that an evaluation of well-typed term does not destroy any live cell in H.

4 Compilation and Type Inference Algorithm

The type system of Λ^{CL} is regarded as a refinement of that of Λ^L with use attributes. Due to this refinement, the former can express efficient list representation which is not achieved in Λ^L. Except for existence of compact list representation, these two type systems are almost equivalent. As we will show later, well-typed terms in Λ^{CL} can be also well-typed in Λ^L by removing all use attribute annotations. The converse also holds, because any term in Λ^L can be considered Λ^{CL} term which always uses usual list representation. One simple consequence of this is that we can obtain a trivial compilation algorithm from Λ^L to Λ^{CL} by directly applying a type inference algorithm for Λ^L. However, such compilation does not give performance

$$\begin{array}{rcl}
\mathit{erase}(\texttt{Nil}) & = & \texttt{Nil} \\
\mathit{erase}(\texttt{Cons}_{\mathit{0}}(M_1,M_2)) & = & \texttt{Cons}(\mathit{erase}(M_1),\mathit{erase}(M_2)) \\
\mathit{erase}(\texttt{Cons}_{\mathit{1}}(M_1,M_2)) & = & \texttt{Cons}(\mathit{erase}(M_1),\mathit{erase}(M_2)) \\
\multicolumn{3}{l}{\mathit{erase}(\texttt{Case } M_{\mathit{1}} \texttt{ of Nil} \Rightarrow M_{\mathit{2}},\ \texttt{Cons}_{\mathit{0}}(x,y) \parallel \texttt{Cons}_{\mathit{1}}(x,y) \Rightarrow M_{\mathit{3}})} \\
 & = & \texttt{Case } \mathit{erase}(M_{\mathit{1}}) \texttt{ of Nil} \Rightarrow \mathit{erase}(M_{\mathit{2}}),\ \texttt{Cons}(x,y) \Rightarrow \mathit{erase}(M_{\mathit{3}})
\end{array}$$

Figure 6: Erasing annotations concerning compact list representation

$$\begin{array}{rcl}
(b^u)^* & = & b \\
((\sigma_1 \to \sigma_2)^u)^* & = & (\sigma_1)^* \to (\sigma_2)^* \\
((\sigma_1\ \texttt{list})^u)^* & = & (\sigma_1)^*\ \texttt{list}
\end{array}$$

Figure 7: Deleting use attribute annotations from type expressions.

improvement. In this section, we present a compilation algorithm which takes account of compact list representation. This algorithm is developed as an extension of Milner's type inference algorithm $\mathcal{W}$ with use attributes.

The first step is to extend the unification algorithm with attributes. We define the algorithm `unify` in the style of [5] by a set of transformation rules on a pair of a set E of type equations and a set S of substitutions. Its transformation terminates if E becomes empty set. Figure 8 gives the set of transformation rules. Using this transformation rules, the function `unify` , which returns a unifier S for a set of type equations E, is defined as follows:

$$\texttt{unify } (E) = \begin{cases} S & \text{(if the transformation } (E,\phi) \text{ terminates with } (\phi,S)) \\ \mathit{error} & \text{(otherwise)} \end{cases}$$

The correctness and termination of this algorithm can be shown similarly to the corresponding proof in [5] for ordinary unification algorithm.

The compilation and type inference algorithm is given in the following form:

$$\mathcal{W}(M) = (\Gamma, M', \tau^u)$$

which takes source term M as an argument, and returns an environment Γ, a term M' in Λ^{CL}, and its type τ^u. The definition of this algorithm is given in Figure 9. In this algorithm, v is a variable for use attributes.

This algorithm infers constraints that v must be $\mathit{0}$. Additional constraints are introduced when two or more sub terms are combined. If a variable occurs in environments returned by recursive calls of the algorithm, and it is given v in more than one of those environments, the algorithm selects one of them as v, and instantiates all the other to $\mathit{0}$. The selection is done by an operator $\oplus_S$. In the definition of this, variables in Γ_2 are always selected. However, many strategies exist for selection. When this algorithm terminates, v in returned components may be instantiated to $\mathit{1}$.

For this inference algorithm, we have the following soundness theorem. *erase* and $(\sigma)^*$ for Λ^{CL} term are defined in Figure 6 and 7 on the structure of terms and types of Λ^{CL} respectively. In Figure 6, trivial cases are omitted.

Theorem 2 *Let M is a well-typed term in Λ^L. If $\mathcal{W}(M) = (\Gamma, M', \tau^u)$, and for all S which substitutes $\mathit{0}$ or $\mathit{1}$ for v and a type for t, then*

- $(E \cup \{(\tau, \tau)\}, S) \Rightarrow (E, S)$
- $(E \cup \{(t, \tau)\}, S) \Rightarrow ([\tau/t]E, [\tau/t]S \cup \{(t, \tau)\})$
- $(E \cup \{(\sigma_{11} \to \sigma_{12}, \sigma_{21} \to \sigma_{22})\}, S) \Rightarrow ((E \cup \{(\sigma_{11}, \sigma_{21}), (\sigma_{12}, \sigma_{22})\}), S)$
- $(E \cup \{(\sigma_1\ \texttt{list}, \sigma_2\ \texttt{list})\}, S) \Rightarrow (E \cup \{(\sigma_1, \sigma_2)\}, S)$
- $(E \cup \{(\tau_1^u, \tau_2^u)\}, S) \Rightarrow (E \cup \{(\tau_1, \tau_2)\}, S)$
- $(E \cup \{(\tau_1^v, \tau_2^u)\}, S) \Rightarrow ([u/v](E \cup \{(\tau_1, \tau_2)\}), [u/v]S \cup \{(v, u)\})$

Figure 8: Unification algorithm

- $S(\Gamma) \rhd S(M') : S(\tau^u)$ *in* Λ^{CL}.
- $erase(M') = M$, *and* $(S(\Gamma))^* \rhd M : (S(\tau^u))^*$ *in* Λ^L.

This theorem states that v in the result of inference may be instantiated to any attribute. We instantiate them to *1* to obtain the compile term. The obtained term is one of optimal ones which our type system infers in the sense that occurrences of *1* is maximized.

As a simple example of the type inference, we consider following `map` function.

$$\lambda g.\texttt{fix}\ f\ x.\texttt{Case}\ x\ \texttt{of}\ \texttt{Nil} \Rightarrow \texttt{Nil},\ \texttt{Cons}(y, z) \Rightarrow \texttt{Cons}(g\ y, f\ z)$$

This function returns a list. Our type inference algorithm infers attributes of the returned list and `Cons` in the second case as v as follows.

$$\lambda g.\texttt{fix}\ f\ x.\texttt{Case}\ x\ \texttt{of}\ \texttt{Nil} \Rightarrow \texttt{Nil},\ \texttt{Cons}_{\mathit{0}}(y, z) \parallel \texttt{Cons}_{\mathit{1}}(y, z) \Rightarrow \texttt{Cons}_v(g\ y, f\ z)$$

This v is then instantiated to *1* safely. As a result, we obtain the following function, which returns a list in compact representation.

$$\lambda g.\texttt{fix}\ f\ x.\texttt{Case}\ x\ \texttt{of}\ \texttt{Nil} \Rightarrow \texttt{Nil},\ \texttt{Cons}_{\mathit{0}}(y, z) \parallel \texttt{Cons}_{\mathit{1}}(y, z) \Rightarrow \texttt{Cons}_{\mathit{1}}(g\ y, f\ z)$$

5 Implementation

5.1 Allocation and Boundary Checking

In our method, a list can be compacted to a consecutive block of arbitrary length. However, in actual implementation, a list block can be allocated only in fixed and finite length. One practical solution is to use the heap space from both ends: from one end we allocate fixed size list blocks, and from the other end we allocate all the other data structures.

A list cell consists of one tag word, which describes its contents, and several data fields. The tag distinguishes lists from any other data structures for garbage collection. A list block is arranged as a sequence of these cells, as shown in Figure 1 at the beginning of this paper. In this runtime representation, since each cell has its own tag word, if there is a pointer to the middle of a block, the pointed cell is recognized as a normal list cell.

With a fixed size list block, a runtime system needs boundary checking every time compact-cons operation is executed. If the boundary checking fails, new list block needs to be allocated.

$\mathcal{W}(c^b) = (\phi, c^b, b^0)$
$\mathcal{W}(x) = (\{x : t^v\}, x, t^v)$ (v is fresh)
$\mathcal{W}(\texttt{fix}\ f\ x.M) =$ let
$(\Gamma, M', \sigma) = \mathcal{W}(M)$
$S_1 =$ substitution which replaces all v given for variables in $\Gamma - \{f, x\}$ for 0
$S_2 = \texttt{unify}\ (S_1(\Gamma(f)), S_1((\Gamma(x) \to \sigma)^0))$
in
$(S_2S_1(\Gamma - \{f, x\}), S_2S_1(\texttt{fix}\ f\ x.M), S_2S_1((\Gamma(x) \to \sigma)^0))$
$\mathcal{W}(M_1M_2) =$ let
$(\Gamma_1, M_1', \sigma_1) = \mathcal{W}(M_1)$
$(\Gamma_2, M_2', \sigma_2) = \mathcal{W}(M_2)$
$S_1 = \Gamma_1 \oplus_S \Gamma_2$
$S_2 = \texttt{unify}\ ((S_1(\Gamma_1) \oplus_{unify} S_1(\Gamma_2)) \cup \{(S_1(\sigma_1), S_1((\sigma_2 \to t^v)^0))\})$ (v is fresh)
in
$(S_2S_1(\Gamma_1) \oplus S_2S_1(\Gamma_2), S_2S_1(M_1'M_2'), S_2(t^v))$
$\mathcal{W}(\texttt{Nil}) = (\phi, \texttt{Nil}, (t^0\ \texttt{list})^v)$ (v is fresh)
$\mathcal{W}(\texttt{Cons}(M_1, M_2)) =$ let
$(\Gamma_1, M_1', \sigma_1) = \mathcal{W}(M_1)$
$(\Gamma_2, M_2', \sigma_2) = \mathcal{W}(M_2)$
$S_1 = \Gamma_1 \oplus_S \Gamma_2$
$S_2 = \texttt{unify}\ ((S_1(\Gamma_1) \oplus_{unify} S_1(\Gamma_2)) \cup \{(S_1(\sigma_2), (S_1(\sigma_1)\ \texttt{list})^v)\})$ (v is fresh)
in
$(S_2S_1(\Gamma_1) \oplus S_2S_1(\Gamma_2), \texttt{Cons}_{S_2(v)}(S_2S_1(M_1'), S_2S_1(M_2')), ((S_2(\sigma_1))\ \texttt{list})^{v'})$ (v' is fresh)
$\mathcal{W}(\texttt{Case}\ M_1\ \texttt{of Nil} \Rightarrow M_2,\ \texttt{Cons}(x, y) \Rightarrow M_3) =$
let
$(\Gamma_1, M_1', \sigma_1) = \mathcal{W}(M_1)$
$(\Gamma_2, M_2', \sigma_2) = \mathcal{W}(M_2)$
$(\Gamma_3, M_3', \sigma_3) = \mathcal{W}(M_3)$
$\Gamma_2' = \Gamma_2 \cup \{x : t^0, y : (t^0\ \texttt{list})^0\}$
$S_1 = \Gamma_2' \oplus_{S'} \Gamma_3$
$S_2 = \texttt{unify}\ ((\Gamma_2' \oplus_{unify} \Gamma_3) \cup \{(\sigma_2, \sigma_3), ((t^0\ \texttt{list})^0, \sigma_1)\})$
$\Gamma_4 = S_2S_1(\Gamma_2) \oplus S_2S_1(\Gamma_3 - \{x, y\})$
$S_3 = \Gamma_1 \oplus_S \Gamma_4$
$S_4 = \texttt{unify}\ ((S_3(\Gamma_1)) \oplus_{unify} S_3(\Gamma_4)$
in
$(S_4S_3(\Gamma_1) \oplus S_4S_3(\Gamma_4)),$
$S_4S_3S_2S_1(\texttt{Case}\ M_1'\ \texttt{of Nil} \Rightarrow M_2',\ \texttt{Cons}_0(x, y) \parallel \texttt{Cons}_1(x, y) \Rightarrow M_3'), S_4S_3S_2S_1(\sigma_3))$

$$\Gamma_1 \oplus_S \Gamma_2 = \{0/v_2 | x \in (Dom(\Gamma_1) \cap Dom(\Gamma_2)), \Gamma_1(x) = \tau_1^{v_1}, \Gamma_2(x) = \tau_2^{v_2}\}$$

$$\Gamma_1 \oplus_{S'} \Gamma_2 = \{v_1/v_2 | x \in (Dom(\Gamma_1) \cap Dom(\Gamma_2)), \Gamma_1(x) = \tau_1^{v_1}, \Gamma_2(x) = \tau_2^{v_2}\}$$

$$\Gamma_1 \oplus_{unify} \Gamma_2 = \{(\tau_1, \tau_2) | x \in (Dom(\Gamma_1) \cap Dom(\Gamma_2)), \Gamma_1(x) = \tau_1^{u_1}, \Gamma_2(x) = \tau_2^{u_2}\}.$$

$$\Gamma_1 \oplus \Gamma_2 = \left\{ \tau^u \,\middle|\, x \in (Dom(\Gamma_1) \cup Dom(\Gamma_2)), \tau^u = \left\{ \begin{array}{ll} \tau^{u_1+u_2} & (x \in Dom(\Gamma_1) \cap Dom(\Gamma_2), \Gamma_i(x) = \tau^{u_i}) \\ \Gamma_1(x) & (x \in Dom(\Gamma_1), x \notin Dom(\Gamma_2)) \\ \Gamma_2(x) & (x \in Dom(\Gamma_2), x \notin Dom(\Gamma_1)) \end{array} \right. \right\}$$

Figure 9: Type Inference Algorithm

To minimize this boundary checking overhead, we allocate a block of size 2^k bytes for some k, so that its boundary is always aligned to 2^k bytes. Boundary checking can, then, be done by checking last k bits in its address. Note that heap boundary checking is required in usual implementations, every time a memory block is allocated.

5.2 Garbage Collection

We consider, here, an issue in adapting a copying garbage collection which is widely used, to our method. When a conventional garbage collector copies a block to "to space", the initial word of the source block is overwritten with a pointer, called "forwarding pointer". However, in our representation, a pointer to the middle of a list block exists. If such a pointer is traced, a forwarding pointer destroys a value in the middle of the block. If a pointer to the beginning of the block is traced later, a block with destroyed cells is incorrectly copied.

Our solution to this problem is outlined as follows. Our garbage collector copies list blocks to "to space" and then process them by scanning "to space" as a conventional algorithm does. To process compact list representation correctly, it maintains the following status information for each copied list cell.

- **garbage:** this cell may a garbage.
- **unprocessed:** this cell is not garbage, and may contain untraced pointers. This is the status for a cell already copied but not scanned.
- **processed:** this cell is not garbage, and is already processed by the garbage collector.

When the collector is given a pointer to a list cell, it processes the cell depending on the status of the block containing given cell, as follows:

- **if the block has not yet copied:** It copies entire block to "to space" and, in the copied block, changes the status of cells in the list (i.e. all those cells from the pointed cell to the end of the block) to "unprocessed" and the status of cells not in the list to "garbage". Then, it writes forwarding pointers to corresponding destinations in all the cells of the source block.
- **if the block has already been copied:** It traces the forwarding pointer of the cell. If the pointed cell exists in the already scanned region, then it processes the block depending on the status of the pointed cell, as follows.
  - **"garbage":** It sets the status of all the "garbage" cells in the list to "processed", then processes them.
  - **"unprocessed":** This case does not occur.
  - **"processed":** It does nothing.

  Otherwise, it sets the status of all the "garbage" cells in the list to "unprocessed".

5.3 Towards a Practical Compiler

In this section, we report results of a preliminary test on the effectiveness of our method. We implemented a translator to abstract machine code designed for this test, and measured execution time of its interpretation. Results are shown in the table below.

benchmark	execution time(sec) usual	compact	Ocaml-2.00	improvement
map	3.73	3.39	1.54	9.35%
reverse	3.96	3.43	5.57	13.4%
append	4.40	4.15	2.44	5.68%
qsort	8.56	8.09	8.31	5.49%

Results under the usual representation and the compact representation are shown in its first and second column, respectively. For reference, in its third column, execution time for the same programs in Ocaml-2.00 [11] is shown. These results show about 5% to 13% improvement in execution time in spite of the existence of significant overheads in interpretation of abstract machine codes. With a native code compiler, more improvement in execution time is expected.

6 Conclusion

We have developed a type-directed list compaction method, which realizes vector-style list representation without increasing runtime overheads in statically typed languages. In our compact representation, a list can be accessed sequentially with simple pointer arithmetic. To achieve this, we have defined a compilation target language with an imperative compact list construction operator. We have, then, developed a type system for this language which allow us to exploit such imperative feature without destroying functional semantics. We also have developed a compilation algorithm from a source language with ordinal list representation to our target language. We have extended a conventional copying garbage collection to our method. Our preliminary experiment shows approximately 5-13% improvement in execution time by our method, in spite of overheads of abstract machine code interpretation. We expect that a native code compiler incorporating the method presented in this paper will achieve a significant performance improvements in list processing.

Acknowledgments

The author would like to thank Atsushi Ohori for his many comments and suggestions.

References

[1] Henry G. Baker. List processing in real time on a serial computer. *Communication of ACM*, 18(7):413–415, July 1978.

[2] Alan Bawden, Richard Greenblatt, Jack Holloway, Thomas Knight, David Moon, and Daniel Weinreb. LISP machine progress report. Technical Memo AIM-444, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, August 1977.

[3] D. G. Bobrow and D. W. Clark. Compact encoding of list structures. *ACM Transactions on Programming Languages and Systems*, 1(2):266–286, October 1979.

[4] L. Peter Deutsch. A LISP machine with very compact programs. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, page 697, Standford, CA, August 1973. William Kaufmann.

[5] J. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67(2):203–260, 1989.

[6] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50, 1987.

[7] Cordelia V. Hall. Using hindley milner type inference to optimise list representation. In *Conference on Lisp and Functional programming*, 1994.

[8] Wilfred J. Hansen. Compact list representation: Definition, garbage collection, and system implementation. *Communications of the ACM*, 12(9):499–507, September 1969.

[9] Naoki Kobayashi. Quasi-linear types. Technical Report TR98-02, Department of Information Science, University of Tokyo, 1998.

[10] Xavier Leroy. Polymorphic typing of an algorithmic language. Technical Report RR-1778, INRIA, 1992.

[11] Xavier Leroy. *The Objective Caml system release 2.01 documentation and user's manual.* INRIA, January 1999.

[12] Kai Li and Paul Hudak. A new list compaction method. *Software — Practice and Experience*, 16(2):145–163, February 1986.

[13] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[14] Yasuhiko Minamide. A functional representation of data structures with a hole. In *The 25rd Symposium on Principles of Programming Languages*, 1998.

[15] Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling lists. In *Conference on Lisp and Functional programming*, June 1994.

[16] David N Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Functional Programming Languages and Computer Architecture*, San Diego, California, 1995.