Unrolling Lists

Zhong Shao*
Princeton University

John H. Reppy†
AT&T Bell Laboratories

Andrew W. Appel*
Princeton University

Abstract

Lists are ubiquitous in functional programs, thus supporting lists efficiently is a major concern to compiler writers for functional languages. Lists are normally represented as linked *cons* cells, with each *cons* cell containing a *car* (the data) and a *cdr* (the link); this is inefficient in the use of space, because 50% of the storage is used for links. Loops and recursions on lists are slow on modern machines because of the long chains of control dependences (in checking for *nil*) and data dependences (in fetching *cdr* fields).

We present a data structure for "unrolled lists," where each cell has several data items (*car* fields) and one link (*cdr*). This reduces the memory used for links, and it significantly shortens the length of control-dependence and data-dependence chains in operations on lists.

We further present an efficient compile-time analysis that transforms programs written for "ordinary" lists into programs on unrolled lists. The use of our new representation requires no change to existing programs.

We sketch the proof of soundness of our analysis—which is based on *refinement types*—and present some preliminary measurements of our technique.

1 Introduction

Efficient implementation of lists has always been a major concern to compiler writers for functional languages, because they occur so frequently in functional programs. Lists are normally represented as linked *cons* cells, with each *cons* cell represented by two contiguous memory locations, one for the *car* (the data) and another for the *cdr* (the link). This is inefficient in the use of space because 50% of the storage is used for links. Furthermore, traversing a list requires twice as many memory references as traversing a vector. And on any loop or recursion that traverses a list, there is a long chain of control dependences as each *cdr* is checked for *nil*; and a long chain of data dependences as each *cdr* fetch is dependent on the previous one. With modern superscalar hardware, these dependences are a serious bottleneck.

In order to save on storage for links, "cdr-coding" was proposed in the 1970's [15, 13, 8, 9, 6, 5]. Its main idea is to try to avoid some links by arranging for the second cons cell to directly follow the car of the first, and to encode that information in several bits contained in the car field of the first cell; thus the first cell does not need a cdr field at all. A depth-first (or breadth-first [4]) copying garbage collector helps ensure that most lists are arranged sequentially in storage, so they can take advantage of this encoding. Cdr-coding solves the space-usage problem (and in the MIT version allows random access subscripting of lists [13]), but makes the control-dependence problem even worse, as the cdr-coding tag of each *car* must be checked. Cdr-coding was popular on microcoded Lisp machines *circa* 1980 [25, 10], but it is not an attractive solution on modern machines.

Our new "compile-time cdr-coding" method works for statically typed languages such as ML. Our scheme allows a more compact runtime representation for lists, but *does not require any runtime encoding at all.* Furthermore, our encoding allows loops and recursions on lists to be unrolled much more efficiently than is possible with the conventional representation for lists.

Our idea is simple: we put k items—but only one link—in each list cell. We use $k = 2$ to illustrate our idea. Lists of *even* length are simply represented as linked series of our bigger cons cells; lists of *odd* length are represented as a header cell that contains one data element and one link to an even-length list. Table 1 gives a simple comparison of space usage between our *new unrolled representation* (NUR) and the *old standard representation* (OSR). In the table, "'a" represents the data element; "0" and "E" represent the tag word that is used to distinguish between odd-length and even-length lists at runtime. We represent the empty list by "0." Now we can easily see that for lists with length greater than 2, the new representation requires 25% less space than the usual representation. Furthermore, traversing a list in the new representation requires 25% fewer loads, and 50% fewer tests for *nil* on cdr pointers (because NUR has 50% fewer cdr links).

The new unrolled representation (NUR) promises to be extremely useful for superscalar or superpipelined machines. Suppose we use a representation with k items per link. Then

*Address: Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544-2087. Email addresses: zsh@cs.princeton.edu and appel@cs.princeton.edu.
The work of these authors was supported by the National Science Foundation Grant CCR-9002786 and CCR-9200790.

†Address: AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974. Email address: jhr@research.att.com.

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

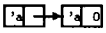
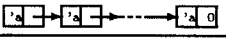
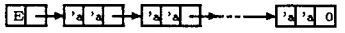
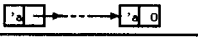
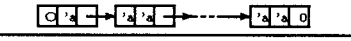
Length	Old Standard Representation	Size	New Unrolled Representation	Size
0	0	0		0[1]
1		2		3
2		4		5
2n		4n		3n+2
2n+1		4n+2		3n+3

Table 1: Comparisons between standard and unrolled list representations

we can unroll most loops on lists by a factor of k, and overlap (using standard software pipelining techniques) the executions of the (original) iterations. Such unrolling and software pipelining would be much less fruitful if performed on the standard list representation (OSR) for two reasons: the tests for nil introduce a chain of $k-1$ extra control dependencies, and the fetches of cdr introduce a chain of $k-1$ extra memory latencies. These chains are a serious obstacle to the software pipelining of anything at all! Note that the fetches of the k *car* fields (in an unrolled loop using NUR) can all be done in parallel; this is not possible in the standard representation.

Because programmers will still use the standard list notation (i.e., each cons cell has one car and one cdr), the compiler has to do the appropriate translation to utilize the new unrolled representations. This is possible in a statically typed language such as ML, because the type of each identifier is statically known at compile time, and computation on lists is expressed using pattern matching and recursive functions. For example, an integer list[2] in ML might be conceptually represented by the following concrete datatype, which matches the standard representation (OSR) in Table 1:

```
datatype list = nil | :: of int * list
```

where "::" is the infix *cons* constructor. The well-known function `map` might be written as follows using pattern matching:

```
fun map f =
  let fun m nil = nil
        | m (x::r) = (f x) :: (m r)
   in m
  end
```

The new unrolled representation (NUR) in Table 1 can also be expressed by ML concrete datatypes:

```
datatype list2 = OLIST of int * tail2
               | ELIST of tail2

     and tail2 = TNIL
               | TAIL2 of int * int * tail2
```

Here, the data constructors `OLIST` and `ELIST` can be thought of as tags for lists of even length and odd length; they correspond to "O" and "E" in Table 1.

An efficient `map` function on NUR lists looks like:

```
fun map' f =
   let fun h (OLIST(i,r)) = OLIST(f i, m r)
           | h (ELIST(r)) = ELIST(m r)
       and m TNIL = TNIL
         | m (TAIL2(x,y,r)) = TAIL2(f x, f y, m r)
    in h
   end
```

The test for *nil* (in the pattern-matching for `m`) is done half as often. If `map'` and then `f` are in-line expanded, then the evaluations of `f x` and `f y` can be overlapped.

Simply unrolling the original `map` function, without changing the list representation, is not as attractive because of the extra control and data dependence:

```
fun map_unrolled f =
  let fun m nil = nil
        | m (x::r) =
            case r
             of nil => (f x) :: nil
              | y::s => (f x) :: (f y) :: (m s)
   in m
  end
```

Our static "list unrolling" transformation has the following advantages over the traditional representation, and over runtime "cdr-coding" techniques:

- Loops and recursions on lists can be efficiently unrolled.
- Even on "sequential" machines, we avoid many *nil* tests and *cdr* fetches.
- Less memory is used for storing links.
- There is no extra runtime cost (as is incurred by cdr-coding) for handling of encoding bits.
- Unlike the "cdr-coding" technique that varies with the dynamic behavior of the program (i.e., cons cells have to be adjacent), our method guarantees a $(k-1)/2k$ savings of space usage for long list structures, using k-fold unrolling.
- The interface with garbage collectors is extremely simple, since we use ordinary record structures.
- Because our transformation only relies on the static type information that is usually available in module interfaces, it interacts very well with the module system and separate compilation.

[1] The two-word record representing the NUR empty list () can be shared among all uses of the empty list. This sharing can be introduced by the garbage collector to avoid complicating the compiled code, if necessary.

[2] In Standard ML [19], lists are declared as `datatype 'a list = nil | :: of 'a * 'a list`. To simplify the presentation, we omit the type variable `'a` by considering only integer lists. All the results described in this paper easily carry to the polymorphic case.

2 Compiling with Refinement Types

In this section, we formally describe the compile-time analysis and present the translation algorithm that automatically transforms program written in OSR notations into one that uses NUR.

First we describe a simple syntactic transformation that gets us partway to our goal. A simple way to implement the NUR is to make the compiler interpret the normal "::" constructor *abstractly*, just as Aitken and Reppy deal with their *abstract value constructors* [1]. During the compilation, the *constructor function* of "::", which takes a data element and a list, and returns a list (the "cons" of the two), can be implemented as the following function ucons:

```
fun ucons(x, OLIST (i,r)) = ELIST (TAIL2 (x, i, r))
  | ucons(x, ELIST r) = OLIST (x, r)
```

The *deconstructor function* (also called *projection*) of "::", which takes a non-empty list, and returns the head and the tail of the list, can be implemented as the following function uproj:

```
fun uproj (OLIST (i,r)) = (i, ELIST r)
  | uproj (ELIST (TAIL2 (i,j,r))) = (i, OLIST (j, r))
```

This approach is extremely easy to implement in most compilers. But it can cause two kinds of runtime inefficiencies when traversing or building a list (such as the map function):

- Both ucons and uproj need to check the length parity of a list each time they are applied, while the old "::" requires no check.
- To build a list using ucons, one must alternately allocate an OLIST cell (e.g., OLIST(j,r)) on the heap, discard an ELIST cell, then take out j and r, build an ELIST cons cell (e.g., ELIST(TAIL2(i,j,r))), and discard the OLIST. This is more expensive than the traditional *cons* operation, which just requires allocating a two element record.

Ideally, the NUR version should avoid the list length parity checks and the alternative allocations of OLIST and ELIST cells, thus be more space and time efficient than the OSR version. The function map' shown in the previous section behaves this way: it first checks whether the argument is of even length or odd length, then the body m of the code "knows" the length parity of its argument.

Now we present a source-to-source program transformation that indeed translates the OSR version of map to this more efficient version map'. The basic idea is to rely on static analysis to distinguish between lists of even length and odd length at compile time, and to allow functions that take lists as arguments to have three entry points: one dispatch function for list whose length parity is unknown, and one specialized version each for list of even length and odd length. Because the specialized versions have the knowledge of the length parity information, the extra runtime costs of the ucons and uproj operations can be avoided.

We can keep track of length parity information for most program variables at compile time, in statically-typed languages such as ML, because lists are accessed via data constructors and pattern matching only, and they are immune to side-effects.[3] We borrow the *refinement type* inference algorithm of Freeman and Pfenning [12, 11] by introducing a *refinement* of the list type: the type olist for odd-length lists and the type elist for even-length lists. For example, an empty list is an even-length list; "consing" an element onto an even-length list yields an odd-length list, and "consing" an element onto an odd-length list yields an even-length list. The map function always returns a list that has the same length parity as its argument list; "append-ing" two lists of same length parity results in an even-length list, and "append-ing" two lists of opposite length parity gives an odd-length list, etc.

In the following, we first define the source language (SRC) that uses the traditional OSR notation and the target language (TGT) that uses the NUR representations. Then we presents an one-pass translation algorithm that infers the length parity information while at the same time compiling SRC expressions into TGT expressions. Finally we sketch the correctness proof method and state the main theorems.

2.1 The source language SRC and the target language TGT

Figure 1 gives the syntax of expressions (ranged over by e), declarations (d), matches (m), and patterns (p) for the source language SRC and the target language TGT. We use c to denote constants, x for program variables, and keywords are underlined. The declarations inside a fix expression may be mutually recursive functions. For the source language, the data constructors nil and :: under OSR are denoted by NIL and CONS in expressions, and by NILP and CONSP in patterns. For the target language, the data constructors under NUR are denoted by OLIST, ELIST, TNIL, TAIL1 and TAIL2 in expressions, and by OLISTP, ELISTP, TNILP, TAIL1P and TAIL2P in patterns. The underlying datatype definition for the NUR version of lists in TGT can be written in ML as follows:

```
datatype list = OLIST of olist
              | ELIST of elist

    and olist = TAIL1 of int * elist

    and elist = TNIL
              | TAIL2 of int * int * elist
```

This is essentially same as the list2 and tail2 type defined in Section 1. The constructor TAIL1 and TAIL1P are introduced to avoid dealing with tuple expressions in our toy language.[4]

The source language SRC can be thought as a typed intermediate language typical of those used in many compilers. Variables and constants are annotated with types, as in x^τ and c^τ. We assume that the SRC programs are typed using the following very simple (monomorphic) types:

$$\tau ::= \iota \mid \texttt{list} \mid \tau_1 \rightarrow \tau_2$$

where ι denotes base types. This does not mean that our algorithm cannot be applied to polymorphic languages; polymorphic expressions can be easily translated

[3] Unlike dynamically typed languages such as Lisp and Scheme, there is no "setcdr" operator in ML.

[4] In practice, TAIL1 is a *transparent* data constructor, thus does not require any extra storage to represent [7, 2].

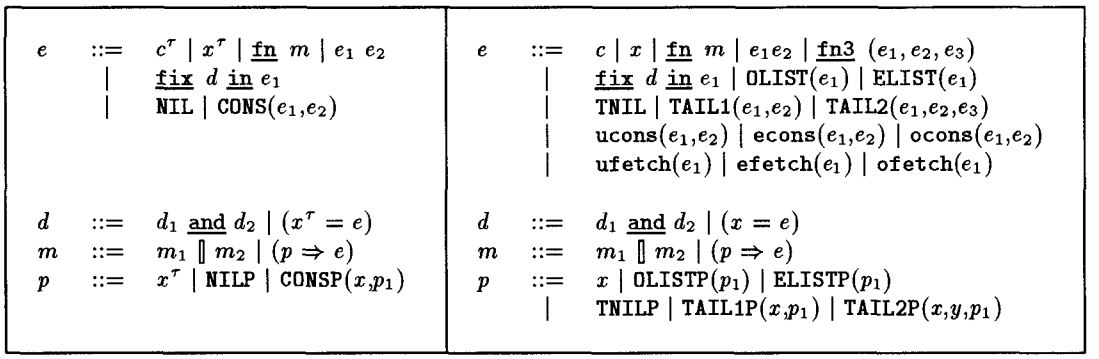


Figure 1: *left:* The Source Language SRC; *right:* The Target Language TGT

into monomorphically-typed intermediate language by using *representation analysis*, a technique first proposed by Leroy [17] and Peyton Jones [21]. Because of space limitations, we have also made several other simplifications to ease the presentation:

- We use integer lists instead of polymorphic lists (but our results easily extend to polymorphic lists).
- We assume that the SRC programs are well-typed according to the standard static typing rules, and that all matches are complete and do not contain redundant patterns.
- Record patterns and expressions are omitted, but pose no problems for our technique.
- Multi-argument functions are also omitted since their translations are similar to translating their curried versions, which are single-argument functions.

The target language TGT has several other constructs and operators: the term fn3 (e_1, e_2, e_3) is used to represent a function that takes a list as argument and has three entry points: one (e_1) for lists whose parity is unknown, and one each (e_2 and e_3) for lists of even length and odd length. The one for unknown parity is always a header function that checks the parity dynamically and immediately dispatches to one of the other two entry points. There are three special operators to extract the appropriate entry point from the term fn3 (e_1, e_2, e_3): `ufetch` to get e_1, `efetch` for e_2, and `ofetch` for e_3. Finally, `ucons` denotes the basic *cons* operation that does not know the length parity of its arguments (the one described at the beginning of this section); `econs` denotes the special operator that *conses* an integer onto an even-length list, and `ocons` *conses* an integer onto an odd-length list. To understand why `econs` and `ocons` can be implemented more efficiently than `ucons`, notice that the expression `ocons`(e_1,`econs`(e_2,e_3)) can be transformed to `TAIL2`(e_1,e_2,e_3), which avoids the parity checking and the allocation of the intermediate odd-length list cell.

2.2 The source-to-target translation

The translation of a source language term into the target language is based on the SRC types and the *refinement types* inferred for the term and its subterms. Our translation proceeds by computing refinement types and the translated term simultaneously. The refinement types we use are defined as follows:

$$\begin{array}{rcl} \rho & ::= & \tau \mid \bot_\tau \mid \texttt{olist} \mid \texttt{elist} \\ & \mid & \tau \rightarrow \rho_1 \mid \langle \texttt{elist} \rightarrow \rho_1, \texttt{olist} \rightarrow \rho_2 \rangle \end{array}$$

where `olist` and `elist` respectively represent even-length and odd-length lists. For every SRC type τ, $\bot_\tau$ denotes its bottom refinement type. We use $\langle \texttt{elist} \rightarrow \rho_1, \texttt{olist} \rightarrow \rho_2 \rangle$ to specially denote a possible refinement type of a SRC type $\texttt{list} \rightarrow \tau$; intuitively, it can be understood as the type of a function that returns ρ_1 when applied to even-length lists, and ρ_2 when applied to odd-length lists. Most of the notations used in this paper is just a simplified version of that used by Freeman and Pfenning [12, 11]: $\rho \sqsubset \tau$ means that the refinement type ρ *refines* the SRC type τ; $\rho_1 \vee \rho_2$ denotes a refinement type that is the *union* of ρ_1 and ρ_2. We formally define our refinement type system in the appendix.

In Figure 2, we present the translation procedures for expressions (ExpComp), declarations (DecComp), matches (MatchComp), and patterns (PatComp) in the source language SRC. The function ExpComp takes a SRC expression e, a substitution S (from SRC program variables to TGT expressions), and a refinement type environment Γ as its arguments; and returns a TGT expression e' and the inferred refinement type ρ for e. Similarly, CompDec takes a SRC declaration d, a substitution S, and a refinement type environment Γ; and it returns the TGT declaration, and the resulting refinement type environment from d. The loop inside CompDec computes the fixed point of the refinement types; this is guaranteed to terminate because there are only finitely many refinement types below any given SRC type (a proof of this is given by Freeman [11, 12]).

Translation of function application (i.e., e_1e_2) is a simple recursive call of ExpComp on e_1 and e_2. Proper coercions must be inserted depending on the inferred refinement types for e_1 and e_2; this is done by the meta operation **applyfun**$(e_1', \rho_1, e_2', \rho_2)$ which is formally defined in the appendix of this paper.

Translation of abstraction (i.e., fn m) is divided into two cases. If the argument is not a list, this is just a simple recursive call to ExpComp.[5] If the argument is a list, the corresponding matches are translated and specialized twice (via MatchComp), once by assuming the argument as an even-length list (i.e., *par* is `elist`), another by assuming the

[5] Since there are no redundant matches, m must have the form fn$(x^\tau \Rightarrow e)$.

ExpComp $(c^\tau, S, \Gamma) = (c, \tau)$
ExpComp $(x^\tau, S, \Gamma) = (S(x), \Gamma(x))$
ExpComp $(\mathtt{NIL}, S, \Gamma) = (\mathtt{TNIL}, \mathtt{elist})$
ExpComp $(\mathtt{CONS}(e_1, e_2), S, \Gamma) =$
  let $(e_1', \mathtt{int}) =$ ExpComp (e_1, S, Γ) and $(e_2', \rho) =$ ExpComp (e_2, S, Γ)
    for $\rho = \mathtt{elist}$, $\mathtt{olist}$, *cons* is respectively **econs** and **ocons**, ρ' is respectively **olist** and **elist**;
    otherwise, *cons* is **ucons** and $\rho' = \rho$;
  in $(cons(e_1', e_2'), \rho)$

ExpComp $(\underline{\mathtt{fn}}\ m^{\mathrm{list}\to\tau}, S, \Gamma) =$
  let $(m_o', \rho_o) =$ MatchComp $(m, S, \Gamma, \mathtt{olist})$ and $(m_e', \rho_e) =$ MatchComp $(m, S, \Gamma, \mathtt{elist})$
    f_u, f_e, f_o be new program variables and $e' = \underline{\mathtt{fn3}}\ (f_u, f_e, f_o)$;
    $d' = (f_u = \mathbf{combine}(f_e, \rho_e, f_o, \rho_o))\ \underline{\mathtt{and}}\ (f_e = \underline{\mathtt{fn}}\ m_e')\ \underline{\mathtt{and}}\ (f_o = \underline{\mathtt{fn}}\ m_o')$
  in $(\underline{\mathtt{fix}}\ d'\ \underline{\mathtt{in}}\ e', \langle \mathtt{elist} \to \rho_e, \mathtt{olist} \to \rho_o \rangle)$

ExpComp $((\underline{\mathtt{fn}}\ (x^\tau \Rightarrow e)), S, \Gamma) =$
  let $(e', \rho) =$ ExpComp $(e, S \pm \{x \mapsto x\}, \Gamma \pm \{x \mapsto \tau\})$
  in $(\underline{\mathtt{fn}}\ (x \Rightarrow e'), \tau \to \rho)$

ExpComp $(e_1 e_2, S, \Gamma) =$
  let $(e_1', \rho_1) =$ ExpComp (e_1, S, Γ) and $(e_2', \rho_2) =$ ExpComp (e_2, S, Γ)
  in $\mathbf{applyfun}(e_1', \rho_1, e_2', \rho_2)$

ExpComp $(\underline{\mathtt{fix}}\ d\ \underline{\mathtt{in}}\ e, S, \Gamma) =$
  let $(d', \Gamma_1) =$ DecComp (d, S, Γ) and $S_1 = \{x \mapsto x \mid x \in \mathrm{Dom}(\Gamma_1)\}$
    $(e', \rho) =$ ExpComp $(e, S \pm S_1, \Gamma \pm \Gamma_1)$
  in $(\underline{\mathtt{fix}}\ d'\ \underline{\mathtt{in}}\ e', \rho)$

DecComp $(d, S, \Gamma) =$
  let assume d is $(x_1^{\tau_1} = e_1)\ \underline{\mathtt{and}}\ \ldots\ \underline{\mathtt{and}}\ (x_k^{\tau_k} = e_k)$; and $\Gamma_{\mathrm{result}} = \{x_i \mapsto \bot_{\tau_i} \mid i = 1, \ldots, k\}$;
    loop $\Gamma_{\mathrm{start}} = \Gamma_{\mathrm{result}}$; $(e_i', \rho_i) =$ ExpComp $(e_i, S, \Gamma \pm \Gamma_{\mathrm{start}})$ where $i = 1, \ldots, k$;
      $\Gamma_{\mathrm{result}} = \{x_i \mapsto \rho_i \mid i = 1, \ldots, k\}$
    until $(\Gamma_{\mathrm{start}} = \Gamma_{\mathrm{result}})$
  in $((x_1 = e_1')\ \underline{\mathtt{and}}\ \ldots\ \underline{\mathtt{and}}\ (x_k = e_k'), \Gamma_{\mathrm{result}})$

MatchComp $(m, S, \Gamma, par) =$
  let Assume $\{p_i \Rightarrow e_i \mid i = 1, \ldots, k\}$ are those rules in "m" that are *compatible* with *par*;
    $(p_i', S_i, \Gamma_i) =$ PatComp (p_i, par) for $i = 1, \ldots, k$
    $(e_i', \rho_i) =$ ExpComp $(e_i, S \pm S_i, \Gamma \pm \Gamma_i)$ for $i = 1, \ldots, k$
    $\rho = \rho_1 \vee \ldots \vee \rho_k$ and $e_i'' = \mathbf{coerce}(e_i', \rho_i, \rho)$ for $i = 1, \ldots, k$
  in $((p_1' \Rightarrow e_1'')[]\ldots[](p_k' \Rightarrow e_k''), \rho)$

PatComp $(\mathtt{NILP}, \mathtt{elist}) = (\mathtt{TNIL}, \emptyset, \emptyset)$;
PatComp $(x, \mathtt{elist}) = (x, \emptyset, \{x \mapsto \mathtt{elist}\})$;
PatComp $(x, \mathtt{olist}) = (\mathtt{TAIL1P}(y, z), \{x \mapsto \mathtt{TAIL1}(y, z)\}, \{x \mapsto \mathtt{olist}\})$ where y,z are new program variables;

PatComp $(\mathtt{CONSP}(x, p), \mathtt{olist}) = (\mathtt{TAIL1P}(y, p'), S \pm \{x \mapsto y\}, \Gamma \pm \{x \mapsto \mathtt{int}\})$
  where $(p', S, \Gamma) =$ PatComp $(p, \mathtt{elist})$ and y is a new program variable;

PatComp $(\mathtt{CONSP}(x, p), \mathtt{elist}) = (\mathtt{TAIL2P}(y, z, p'), S \pm \{x \mapsto y\}, \Gamma \pm \{x \mapsto \mathtt{int}\})$
  where $(\mathtt{TAIL1P}(z, p'), S, \Gamma) =$ PatComp $(p, \mathtt{olist})$ and y be a new program variable.

Note: The special operators **combine**, **applyfun** and **coerce** are formally defined in the appendix.

Figure 2: Translation of Expressions, Matches, Declarations and Patterns

SRC expression $e = \underline{\texttt{fix}}\ (m = \underline{\texttt{fn}}\ (\texttt{NILP} \Rightarrow \texttt{NIL})[](\texttt{CONSP}(x, r) \Rightarrow \texttt{CONS}(x + 1, m\ r)))\ \underline{\texttt{in}}\ m$

TGT expression $e' = \underline{\texttt{fix}}\ m' = d'\ \underline{\texttt{in}}\ m'$ where

$$d' = \underline{\texttt{fix}}\ ((f_u = e_u)\underline{\texttt{and}}(f_e = e_e)\underline{\texttt{and}}(f_o = e_o))\ \underline{\texttt{in}}\ (\underline{\texttt{fn3}}\ (f_u, f_e, f_o));$$
$$e_u = \underline{\texttt{fn}}\ (\texttt{OLISTP(x)} \Rightarrow \texttt{OLIST}(f_o\ x))[](\texttt{ELISTP(y)} \Rightarrow \texttt{ELIST}(f_e\ y));$$
$$e_o = \underline{\texttt{fn}}\ (\texttt{TAIL1P}(x, r) \Rightarrow \texttt{ocons}(x + 1, ((\texttt{efetch}(m'))\ r)));$$
$$e_e = \underline{\texttt{fn}}\ (\texttt{TNILP} \Rightarrow \texttt{TNIL})[](\texttt{TAIL2P}(x, y, r) \Rightarrow \texttt{econs}(x + 1, ((\texttt{ofetch}(m'))\ (\texttt{TAIL1}(y, r)))))$$

Figure 3: Example on the map function

argument as an odd-length list (i.e., *par* is olist); these two resulting TGT matches (f_e, f_o) correspond to two specialized entry points for lists of even length and odd length. The special entry point f_u for lists of unknown length is built by the **combine** operation: **combine**(f_e, ρ_e, f_o, ρ_o) is a TGT function that checks the length parity of its argument first and then dispatch it to special versions f_e or f_o.

The argument *par* in the MatchComp procedure represents the length parity (either elist or olist) of the argument in the match m. A simple SRC rule $p \Rightarrow e$ is *compatible* with *par* if p is *compatible* with *par*. The *compatibility* between a SRC pattern and a parity is inductively defined as follows: variable pattern x is compatible with both elist and olist; NILP is only compatible with elist; CONSP(x, p) is compatible with elist (or olist) if and only if p is compatible with olist (or elist).

The PatComp procedure translates a SRC pattern p into the TGT pattern based on the parity assumption about p; it also derives a substitution and a refinement type environment for all variables in p.

For example, Figure 3 shows the target expression from translating a simplified version of the map function (shown rather than as in Section 1). This function maps the "+1" function to a list. Our algorithm infers that m has the refinement type (elist → elist, olist → olist) and yield the target expression e'. Notice that in the real implementation, the expression efetch(m') (inside e_o) and ofetch(m') (inside e_e) will be contracted into f_e and f_o, and the application of f_o to TAIL1(y, r) (inside e_e) will be inline-expanded, then the consecutive application of econs and ocons in e_e will be contracted into TAIL2($x + 1, y + 1, f_e\ r$). This is exactly the form we desired in Section 1.

2.3 Correctness of the translation

The type and semantic correctness of our translation can be proven using a technique similar to that of Leroy [17]. Because of space limitations, here we only sketch the proof method and state the main theorem. We use $\vdash_{\text{SRC}}$ to denote the type deduction rule for SRC, and $\vdash_{\text{TGT}}$ to denote the refinement type deduction rule for TGT. More specifically, suppose TE is a SRC type environment (from variables to SRC type τ), Γ is a refinement type environment, TE $\vdash_{\text{SRC}}$ $e : \tau$ means that e is well-typed in TE under $\vdash_{\text{SRC}}$, and $\Gamma \vdash_{\text{TGT}} e' : \rho$ means e' has the refinement type ρ in Γ under $\vdash_{\text{TGT}}$. We also define the (straight-forward) *call-by-value* operational semantics VE $\vdash e \xrightarrow{s} v$ for the source language SRC, and VE$'$ $\vdash e' \xrightarrow{t} v'$ for the target language TGT, where VE and VE$'$ are *value environments* (from variables to values). A notion of *equivalence* between the typed SRC *values* (which corresponds to the OSR) and the typed TGT *values* (which corresponds to the NUR) is defined, written as $v : \tau \approx v' : \rho$. This $\approx$ relation is only defined for the pair of values when v has type τ, v' has type ρ, and $\rho \sqsubset \tau$. VE : TE $\approx$ VE$'$: Γ is used to denote that for every $x \in$ Dom(VE), such that VE(x) : TE(x) $\approx$ VE$'$(x) : $\Gamma(x)$. The type and semantic correctness of our translation algorithm now can be stated by the following proposition, which is proven by structural induction:

Proposition 2.1 *Given a* SRC *expression* e, *a* SRC *type environment* TE, *and a refinement type environment* Γ, *such that* TE $\vdash_{\text{SRC}}$ $e : \tau$ *is valid, and* $\Gamma(x) \sqsubset$ TE(x) *for every* $x \in$ Dom(TE), *then* ExpComp $(e, ID, \Gamma) = (e', \rho)$ *will succeed; moreover, (1)* $\rho \sqsubset \tau$; *(2)* $\Gamma \vdash_{\text{TGT}} e' : \rho$ *is valid; (3) Given a value environment* VE *under* $\xrightarrow{s}$ *and a value environment* VE$'$ *under* $\xrightarrow{t}$, *if* VE : TE $\approx$ VE$'$: Γ, *and* VE $\vdash e \xrightarrow{s} v$, *then, there exists a value* v' *such that* VE$'$ $\vdash e' \xrightarrow{t} v'$ *and* $v : \tau \approx v' : \rho$.

3 Compiling with Multiple Continuations

The algorithm presented in Section 2 successfully translates a program written in OSR notation into one in NUR. In most cases, it pleasantly eliminates the costs of extra length parity checks and alternating allocations of OLIST and ELIST cells incurred by ucons and uproj. To demonstrate this, we tried our algorithm on 15 frequently used list-processing library functions.[6] Among these 15 cases, our algorithm successfully eliminates all the extra costs of ucons and uproj for 14 of them. The only exception is the filter function, which selects only those element of a list matching a given predicate. The problem with filter is that even if we know the length parity of its argument, we still do not know the length parity of its result.

Here is the filter function that takes a predicate p and a list, and returns a list of all elements satisfying the predicate p:

```
fun filter p =
  let fun f nil = nil
        | f (x::r) = if (p x) then x::(f r) else f r
   in f
  end
```

[6] Here is a list of these functions: hd, tl, length, append, rev, map, fold, revfold, app, revapp, nthtail, nth, exists, last, and filter. They are mostly taken from the initial basis of the Standard ML of New Jersey compiler [3].

In this case, even we know the length parity of the argument list, there is still no way to know the parity of the result "f r."

It turns out that this problem can be easily solved in the continuation-passing style (CPS) framework [23, 2], because we can specialize the return continuation on the length parity of the result, and make it have multiple entry points also. The idea is as follows: when we are converting a SRC expression e into CPS, we use a method similar to that of Section 2 to infer the refinement type e; whenever we are not sure about the length parity of a list expression, we duplicate its return continuation into one accepting an even-length list and another accepting an odd-length list. For example, the source-language `filter` function is CPS-converted into the `filter'` function in Figure 4 (written using pseudo-CPS notation in ML). Here, `c`, `ce`, `co`, `ke`, `ko` are the continuation

```
fun filter' (p, c) =
  let fun f_u(OLIST(x,r), ce, co) = f_o(x, r, ce, co)
        | f_u(ELIST r, ce, co) = f_e(r, ce, co)

      and f_o(x, r, ce, co) = if (p x)
            then let fun ke(z) = co(econs(x,z))
                     fun ko(z) = ce(ocons(x,z))
                 in f_e(r, ke, ko)
                 end
            else f_e(r, ce, co)

      and f_e(TNIL, ce, co) = ce(TNIL)
        | f_e(TAIL2(x, y, r), ce, co) = if (p x)
            then let fun ke(z) = co(econs(x,z))
                     fun ko(z) = ce(ocons(x,z))
                 in f_o(y, r, ke, ko)
                 end
            else f_o(y, r, ce, co)

   in c(f_u)
  end
```

Figure 4: Pseudo CPS code for `filter`

variables. The length parity of the variable `z` (i.e., the return result of `f_e` and `f_o`) is statically unknown, but after duplicating the return continuation (`ke`, `ko`), `z` is then assumed as even-length list and odd-length list in each, thus no parity check is necessary, and the more efficient version (`econs` and `ocons`) of "ucons" can be used. The heap allocation of intermediate `OLIST` cons cell is still avoided because of representation analysis [17].

It is likely, however, that this transformation will only improve performance if the underlying compiler uses representation analysis [17], and is very sophisticated about closure construction and register usage. Otherwise, the extra cost of closure creations could outweigh the elimination of the cons operations.

Note, however, that though there are extra costs of testing for ELIST/OLIST, there are fewer tests for *nil*. The result (as shown in the next section) is that the NUR version of filter is about as fast as the OSR version, even without the specialized CPS version of our analysis.

4 Experiments

We have implemented the algorithm described in Section 2 in an experimental version of the Standard ML of New Jersey compiler (SML/NJ) [3, 2]. Because the compiler uses continuation-passing style as its intermediate language, the multiple-continuation approach described in Section 3 can be easily added (this has not been done yet). The SML/NJ compiler supports representation analysis [17], so intermediate odd-length lists are represented by unboxed records, which normally stay in registers; this makes the specialized versions (for even-length and odd-length lists) of ucons and `uproj` operations involve even fewer memory allocations.

4.1 Avoiding code explosion

Translating from OSR to NUR involves function specialization and recursion unrolling. If a function takes n list arguments with a k-way unrolled representation, we will need k^n entry points.

Though most list-processing functions take only one list argument, for functions that take multiple list arguments (e.g., the append function which concatenates two lists), an exponential blowup is a serious concern.

To avoid the blowup, we use a system parameter called `unroll-level` to control the depth of specialization and unrolling. For example, suppose function `f` has five arguments that are of type `list`, and suppose `unroll-level` is 2, then the compiler will only specialize the first two arguments, but other three will not be specialized. The slight runtime cost for not specializing some arguments is not a problem in practice because most frequently-used list functions are often one or two argument functions. For example, among the 15 functions in the initial "List" library in the SML/NJ compiler, 14 of them have only one list argument, and only the append function has two list arguments.

4.2 Measurements

Our technique guarantees a 25% savings in memory usage for (long) lists. But execution time savings will be achieved only if most of the ucons and `uproj` operations can be removed.

To demonstrate the savings of execution time, we have compared the performance of several benchmarks under the standard representation (OSR) and the unrolled representation (NUR). Our benchmarks include: **life**, the game of Life implemented using lists (written by Reade [22]); **ray**, a simple ray tracer (this program does not contain much list processing); **quicksort**, sorting a list of 20000 real numbers using the quicksort algorithm (taken from Paulson [20]); **samsort**, sorting a list of 2000 real numbers using a variation of mergesort algorithm (taken from Paulson [20]); **intset**, "set" operations on sets of integers implemented with sorted lists; **mmap**, several runs of the `map` function on a long list.

Table 2 gives the total size of lists allocated during execution, the program execution time on a DECstation 5000/240, and the code size increase, with the `unroll-level` set as 2, and each *cons* cell 2-way unrolled (i.e., k=2). In all cases, the NUR version allocates less[7] and runs faster (11%-25%) than the OSR version. Notice that **life** benchmark frequently

[7]NUR allocates 33% less than OSR on certain benchmarks, because unlike in Table 1, each *cons* cell in our compiler still contains an extra descriptor word.

Benchmark	Lists Allocated (mega-words)			CPU Time (seconds)			Code Size (kilo-bytes)		
	OSR	NUR	Savings	OSR	NUR	Savings	OSR	NUR	Ratio
life	0.71	0.71	0%	1.19	0.91	23%	13.9	54.5	x3.9
ray	13.89	13.89	0%	22.71	20.59	9%	44.0	77.5	x1.76
quicksort	1.81	1.35	25%	0.98	0.87	11%	3.0	8.4	x2.8
samsort	1.81	1.30	29%	1.03	0.88	15%	2.5	5.3	x2.1
intset	0.54	0.36	33%	0.63	0.51	19%	4.1	12.3	x3.0
mmap	1.20	0.80	33%	1.73	1.29	25%	3.5	8.7	x2.5

Table 2: Performance of the Benchmark Programs

calls the `filter` function, and several functions that have more than two list arguments (thus some of them are not specialized); because of this, the total size of lists allocated for NUR is about the same as OSR; but because NUR requires many fewer memory references and *nil* tests, it runs much faster than OSR (about 23%). Although the code size did not explode because of the `unroll-level` parameter, it does increase by a factor of 1.76 to 3.9. We are currently exploring ways of cutting down the code size for NUR, while still maintaining its performance gain. One problem of our current implementation of NUR is that it does not have a good dead code detection algorithm, we believe that a more refined implementation can achieve more code sharing and produce much smaller code.

5 Related Work

Cdr-coding techniques were first proposed in early 70's by a group of researchers at MIT and Xerox [15, 13, 8, 9, 6, 5]. While these schemes differ from each other on the encoding methods, they all rely on the hardware support from microcoded Lisp machines [25, 10] to alleviate the high costs incurred by the runtime encoding bits. Since modern machines tend not to offer these kinds of special hardware support, the runtime cdr-coding technique quickly became obsolete in the 1980's. The "static cdr-coding" technique presented in this paper is a simple compile-time method for doing list compaction. It is attractive for modern machines because it does not require any runtime encoding bits at all.

Li and Hudak [18] proposed a cdr-coding scheme for list compaction under parallel environments. When several lists are being constructed simultaneously from the same heap, the non-contiguous nature of the cells being allocated might eliminate the opportunity for compaction under traditional cdr-coding techniques. To overcome this, they also represent list as linked (fixed length) vectors, and do the "consing" by pre-allocating a vector first and consecutively filling in later elements. This technique still relies on runtime encoding bits to distinguish the state of each vector cell (i.e., filled or empty), and is thus quite expensive. Our static cdr-coding method, on the other hand, exploits compile-time analysis to eliminate most runtime checks; at the same time, it poses no more problem in parallel environments than does "ordinary" consing.

On the side of statically typed languages, Hall [14] has presented a list compaction technique for Haskell [16]. In her scheme, lists can be represented as the old standard representation (OSR) at one place, and in an optimized representation at another place. The optimized representation in her scheme is adapted for lazy languages where the tail of a list may not yet be evaluated, and thus its length parity cannot be known. Therefore, she must put the "extra" elements at the end of the list, making the test on each unrolled iteration more complicated. In part because of this, her scheme requires more runtime checks than ours. Hall's analysis (based on Hindley-Milner type inference) determines where to insert coercions between different representations. But the representations themselves must already be used in the programs. In effect, this means that library functions must be explicitly programmed using several different representations, and programs will be improved only if they use the library functions.

The idea of using special and more efficient representations for frequently used data objects is originally from Leroy [17] and Peyton Jones [21]. Both propose a type-based program transformation scheme that allows objects with monomorphic ML types to use special unboxed representations. When an unboxed object interacts with a boxed polymorphic object, appropriate coercions are inserted. But as mentioned by Leroy [17], their representation analysis techniques do not work well with ML's recursive data types, such as the list type. This is because the coercion between the unboxed and boxed representation for lists is often rather expensive (i.e., has costs proportional to the list length). Our translation scheme, on the other hand, allows commonly used list objects to uniformly use more efficient unrolled representations, whether they have a monomorphic type or not—though the *element values* must still use the general (single-word) representation. Coercions among representations for even-length lists, odd-length lists, and lists whose length parity is unknown, are quite cheap.

The refinement type system used in Section 2 is a much simplified version of Freeman and Pfenning's refinement type system [12, 11]. While the underlying framework and type inference algorithm are quite similar, our motivation is rather different. In their system, the refinement type is declared by the programmer, and the refinement type information is used to detect program errors at compile time. The reason that we use the refinement types, on the other hand, is to do compile-time program transformations and optimizations. The refinement type declaration used in our scheme is embedded in the compiler, and is completely hidden from programmers.

As in Wadler's *views* mechanism [24], the standard and unrolled representations of lists in our scheme can be linked together by a pair of **in** and **out** functions (e.g., the "`ucons`" and "`uproj`" function in Section 1). We introduce unrolled representation for lists mainly to improve the space and time

efficiencies for programs using lists, while Wadler uses his *views* mechanism to hide the representations of concrete data types and reconcile pattern matching with data abstraction.

6 Conclusions

We have presented a "list unrolling" technique that allows a more compact and efficient list representation for statically typed languages. Our "list unrolling " technique generalizes to depth-d unrolling of k-ary trees with only k^{dm} entry points necessary for any function with m tree arguments. Reasoning about lists (and trees) in these languages are easier than on pointers in other languages because lists (and trees) are accessed only via data constructors and pattern matching. The higher-level of language abstraction makes permits the compiler to automatically transform a program into one that uses more efficient data representations, and that permits loop unrolling by eliminating certain control and data dependences.

References

[1] William E. Aitken and John H. Reppy. Abstract value constructors. In *ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[4] Henry G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[5] Daniel G. Bobrow. A note on hash linking. *Communications of the ACM*, 18(7):413–415, July 1975.

[6] Daniel G. Bobrow and Douglas W. Clark. Compact encodings of list structure. *ACM Transactions on Programming Languages and Systems*, 1(2):267–286, October 1979.

[7] Luca Cardelli. Compiling a functional language. In *Proc. of the 1984 ACM Conference on Lisp and Functional Programming*, pages 208–217, August 1984.

[8] Douglas W. Clark. *List structure: measurements, algorithms, and encodings.* PhD thesis, Carnegie Mellon Univ., Pittsburgh, PA, August 1976.

[9] Douglas W. Clark and C. Cordell Green. An empirical study of list structure in lisp. *Communications of the ACM*, 20(2):78–87, February 1977.

[10] L. P. Deutsch. A lisp machine with very compact programs. In *Proc. 3rd IJACI*, pages 697–703, 1973.

[11] Tim Freeman. Carnegie Mellon University, personal communication, 1992.

[12] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proc. ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, pages 268–277, New York, July 1991. ACM Press.

[13] R. Greenblatt. Lisp machine progress report memo 444. Technical report, A.I. Lab., M.I.T., Cambridge, MA, August 1977.

[14] Cordelia V. Hall. Using hindley-milner type inference to optimize list representation. In *1994 ACM Conference on Lisp and Fucntional Programming*, page (to appear), New York, June 1994. ACM Press.

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

[16] Paul Hudak, Simon L. Peyton Jones, and Philip Wadler *et al.* Report on the programming language Haskell a non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 21(5), May 1992.

[17] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1992. ACM Press.

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

[19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[20] Laurence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, Cambridge, 1992.

[21] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *The Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, New York, August 1991. ACM Press.

[22] Chris Reade. *Elements of Functional Programming.* Addison-Wesley, Reading, MA, 1989.

[23] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.

[24] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Fourteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 307–313, New York, Jan 1987. ACM Press.

[25] D. Weinreb and D. Moon. Lisp machine manual. Technical report, Symolics Corp., Cambridge, Mass., 1981.

A Appendix

A.1 An introduction to refinement types

In this section, we give a brief introduction about the *refinement type* system used in Section 2.2. Most of the notations and concepts are directly borrowed from Freeman and Pfenning [12, 11], since our system is just a simplified version of theirs. Basically we *refine* the `list` type by introducing `elist` for even-length lists and `olist` for odd-length lists. Functions that take lists as arguments can have a more "refined" type, $\langle \texttt{elist} \rightarrow \rho_1, \texttt{olist} \rightarrow \rho_2 \rangle$, meaning that the result has type ρ_1 if applied to an even-length list, and ρ_2 if applied to an odd-length list. The refinement types (ranged over by ρ) are formally defined as follows:

$$\begin{array}{rcl} \rho & ::= & \tau \mid \bot_\tau \mid \texttt{olist} \mid \texttt{elist} \mid \tau \rightarrow \rho_1 \\ & \mid & \langle \texttt{elist} \rightarrow \rho_1, \texttt{olist} \rightarrow \rho_2 \rangle \end{array}$$

For every SRC type τ, $\bot_\tau$ represents its bottom refinement type. In the following, we say that a refinement type ρ *refines* an SRC type τ, written $\rho \sqsubset \tau$, if it can be deduced by the rules (R1-R6) in Figure 5. Notice that we only refine the domain of a function type if it is a list type,

We say that a refinement type ρ_1 is a *subtype* of another refinement type ρ_2, written $\rho_1 \leq \rho_2$, if it can be deduced by the rules (S1-S7) in Figure 5. Similarly, Two refinement types ρ_1 and ρ_2 are *equal*, denoted by $\rho_1 \equiv \rho_2$, if $\rho_1 \leq \rho_2$ and $\rho_2 \leq \rho_1$. $\equiv$ is an *equivalence* relation on refinement types.

(R1) $\tau \sqsubset \tau$; (R2) $\bot_\tau \sqsubset \tau$; (R3) $\mathtt{elist} \sqsubset \mathtt{list}$; (R4) $\mathtt{olist} \sqsubset \mathtt{list}$;
(R5) $(\tau \to \rho) \sqsubset (\tau \to \tau')$ if $\rho \sqsubset \tau'$;
(R6) $\langle \mathtt{elist} \to \rho_1, \mathtt{olist} \to \rho_2 \rangle \sqsubset (\mathtt{list} \to \tau')$ if $\rho_1 \sqsubset \tau'$ and $\rho_2 \sqsubset \tau'$.

(S1) $\rho \leq \rho$; (S2) $\rho_1 \leq \rho_3$ if $\rho_1 \leq \rho_2$ and $\rho_2 \leq \rho_3$;
(S3) $\rho \leq \tau$ and $\bot_\tau \leq \rho$ if $\rho \sqsubset \tau$; (S4) $\tau \to \rho_1 \leq \tau \to \rho_2$ if $\rho_1 \leq \rho_2$;
(S5) $\mathtt{list} \to \rho \leq \langle \mathtt{elist} \to \rho_1, \mathtt{olist} \to \rho_2 \rangle$ if $\rho \leq \rho_1$ and $\rho \leq \rho_2$;
(S6) $\langle \mathtt{elist} \to \rho_1, \mathtt{olist} \to \rho_2 \rangle \leq \mathtt{list} \to \rho$ if $\rho_1 \leq \rho$ and $\rho_2 \leq \rho$;
(S7) $\langle \mathtt{elist} \to \rho_1, \mathtt{olist} \to \rho_2 \rangle \leq \langle \mathtt{elist} \to \rho_1', \mathtt{olist} \to \rho_2' \rangle$ if $\rho_1 \leq \rho_1'$ and $\rho_2 \leq \rho_2'$.

(T1) $\mathbf{top}(\tau) = (\tau)$; (T2) $\mathbf{top}(\bot_\tau) = (\tau)$; (T3) $\mathbf{top}(\mathtt{elist}) = (\mathtt{list})$; (T4) $\mathbf{top}(\mathtt{olist}) = (\mathtt{list})$;
(T5) $\mathbf{top}(\tau \to \rho) = \tau \to (\mathbf{top}(\rho))$; (T6) $\mathbf{top}(\langle \mathtt{elist} \to \rho_1, \mathtt{olist} \to \rho_2 \rangle) = \mathtt{list} \to (\mathbf{top}(\rho_1))$.

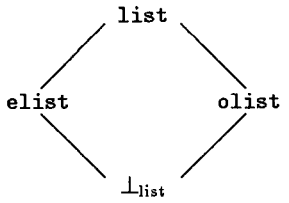
(U1) $\rho \vee \rho' = \rho'$ and $\rho' \vee \rho = \rho'$ if $(\rho \leq \rho')$;
(U2) $(\mathtt{olist}) \vee (\mathtt{elist}) = \mathtt{list}$ and $(\mathtt{elist}) \vee (\mathtt{olist}) = \mathtt{list}$;
(U3) $(\tau \to \rho_1) \vee (\tau \to \rho_2) = \tau \to (\rho_1 \vee \rho_2)$;
(U4) $\langle \mathtt{elist} \to \rho_1, \mathtt{olist} \to \rho_2 \rangle \vee (\mathtt{list} \to \rho) = \langle \mathtt{elist} \to \rho_1 \vee \rho, \mathtt{olist} \to \rho_2 \vee \rho \rangle$;
(U5) $(\mathtt{list} \to \rho) \vee \langle \mathtt{elist} \to \rho_1, \mathtt{olist} \to \rho_2 \rangle = \langle \mathtt{elist} \to \rho_1 \vee \rho, \mathtt{olist} \to \rho_2 \vee \rho \rangle$;
(U6) $\langle \mathtt{elist} \to \rho_1, \mathtt{olist} \to \rho_1' \rangle \vee \langle \mathtt{elist} \to \rho_2, \mathtt{olist} \to \rho_2' \rangle = \langle \mathtt{elist} \to \rho_1 \vee \rho_2, \mathtt{olist} \to \rho_1' \vee \rho_2' \rangle$

(A1) $\mathbf{apprfty}(\bot_{\tau_1 \to \tau_2}, \rho_1) = \bot_{\tau_2}$ if $\rho_1 \sqsubset \tau_1$; (A2) $\mathbf{apprfty}(\tau \to \rho, \bot_\tau) = \bot_{\mathbf{top}(\rho)}$;
(A3) $\mathbf{apprfty}(\tau_1 \to \rho_2, \rho_1) = \rho_2$ if $\rho_1 \sqsubset \tau_1$;
(A4) $\mathbf{apprfty}(\rho, \bot_{\tau\ \mathtt{list}}) = \bot_{\mathbf{top}(\rho_1 \vee \rho_2)}$ and $\mathbf{apprfty}(\rho, \mathtt{list}) = \rho_1 \vee \rho_2$ and
$\mathbf{apprfty}(\rho, \mathtt{elist}) = \rho_1$ and $\mathbf{apprfty}(\rho, \mathtt{olist}) = \rho_2$ where $\rho = \langle \mathtt{elist} \to \rho_1, \mathtt{olist} \to \rho_2 \rangle$.

Figure 5: Definitions of "$\sqsubset$", "$\leq$", "$\vee$", "**top**", and "**apprfty**" on refinement types

In this paper, when we talk about a refinement type ρ, we refer to its equivalence class under $\equiv$.

Every SRC type τ has a finite number of refinement types. Moreover, these refinement types form a lattice under the subtype relation "$\leq$", with τ as its top and $\bot_\tau$ as its bottom. For example, the lattice of refinement types for the SRC type `list` is:



Given a refinement type ρ, it is always possible to find out which SRC type it refines. This is denoted by the **top** operation, which is defined by rules (T1-T5) in Figure 5.

Given two refinement types ρ_1 and ρ_2, if $\mathbf{top}(\rho_1) = \mathbf{top}(\rho_2)$, then their *union* type, written as $\rho_1 \vee \rho_2$, is still a refinement type, and is inductively defined by rules (U1-U6). It is easy to see that given two refinement types ρ_1 and ρ_2, if $\rho = \rho_1 \vee \rho_2$, then $\rho_1 \leq \rho$ and $\rho_2 \leq \rho$ (proven by structural inductions on ρ).

The operation **apprfty** of applying a refinement type ρ_1 to another refinement type ρ_2 is defined by rules (A1-A4) in Figure 5. This operation is used extensively by the meta operation **applyfun** defined in Section A.2.

Freeman and Pfenning [11, 12] give an refinement type inference algorithm for typed core-ML. The inference algorithm we used in Section 2.2 for the language SRC is just an adaption of their algorithm to the above refinement type system.

Notice that given an SRC expression e of type τ, the translation algorithm in Section 2.2 returns the corresponding TGT expression e' and a refinement type ρ for e. This "ρ" happens to be the type of e' under $\vdash_{\mathrm{TGT}}$, where "$\vdash_{\mathrm{TGT}}$" is just a set of (simple monomorphic) typing rules for TGT expressions by ρ defined above extended with "$\mathtt{elist} \to \rho_1$" and "$\mathtt{olist} \to \rho_1$". For example, the typing rule for the **econs** expression will be "if $\Gamma \vdash_{\mathrm{TGT}} e_1 : \mathbf{int}$ and $\Gamma \vdash_{\mathrm{TGT}} e_2 : \mathtt{elist}$; then $\Gamma \vdash_{\mathrm{TGT}} \mathbf{econs}(e_1, e_2) : \mathtt{olist}$;" the rule for <u>fn3</u> (e_1, e_2, e_3) will be "if $\Gamma \vdash_{\mathrm{TGT}} e_1 : \mathtt{list} \to \rho$ and $\Gamma \vdash_{\mathrm{TGT}} e_2 : \mathtt{elist} \to \rho_e$ and $\Gamma \vdash_{\mathrm{TGT}} e_3 : \mathtt{olist} \to \rho_o$ and $\rho = \rho_e \vee \rho_o$; then $\Gamma \vdash_{\mathrm{TGT}}$ <u>fn3</u> $(e_1, e_2, e_3) : \langle \mathtt{elist} \to \rho_e, \mathtt{olist} \to \rho_o \rangle$. These typing rules are straightforward, and are thus omitted here.

A.2 The definition of several meta-operations

In this section, we formally define the three meta operations **coerce**, **combine** and **applyfun** used in the translation algorithm in Figure 2.

Figure 6 gives the definition of **combine**. Given two tar-

$\mathbf{combine}(e'_1, \rho_1, e'_2, \rho_2) = \underline{\mathtt{fn}}\ (\mathtt{OLIST}\ x \Rightarrow \mathbf{coerce}(e'_1 x, \rho_1, \rho))[](\mathtt{ELIST}\ y \Rightarrow \mathbf{coerce}(e'_1 y, \rho_2, \rho))$
where $\rho = (\rho_1 \vee \rho_2)$ and x,y are new program variables

$\mathbf{coerce}(e', \rho, \rho) = e';$
$\mathbf{coerce}(e', \bot_\tau, \rho) = e';$
$\mathbf{coerce}(e', \mathtt{elist}, \mathtt{list}) = \mathtt{ELIST}(e');$
$\mathbf{coerce}(e', \mathtt{olist}, \mathtt{list}) = \mathtt{OLIST}(e');$

$\mathbf{coerce}(e', \tau \rightarrow \rho_1, \tau \rightarrow \rho_2) = \underline{\mathtt{fn}}\ (x \Rightarrow \mathbf{coerce}((e'\ x), \rho_1, \rho_2));$

$\mathbf{coerce}(e', \langle \mathtt{elist} \rightarrow \rho_1, \mathtt{olist} \rightarrow \rho_2 \rangle, \mathtt{list} \rightarrow \rho') =$
$\underline{\mathtt{fn}}\ (x \Rightarrow \mathbf{coerce}(((\mathtt{ufetch}(e'))\ x), \rho_1 \vee \rho_2, \rho'));$

$\mathbf{coerce}(e', \mathtt{list} \rightarrow \rho', \langle \mathtt{elist} \rightarrow \rho_1, \mathtt{olist} \rightarrow \rho_2 \rangle) = \underline{\mathtt{fix}}\ d'\ \underline{\mathtt{in}}\ (\underline{\mathtt{fn3}}\ (f_u, f_e, f_o))$
where f_u, f_e, f_o are new program variables
and $d' = (f_u = m'_u)\ \underline{\mathtt{and}}\ (f_e = m'_e)\ \underline{\mathtt{and}}\ (f_o = m'_o)$ and $m'_u = (\mathbf{combine}(f_e, \rho_1, f_o, \rho_2))$
and $m'_e = (x \Rightarrow \mathbf{coerce}((e'\ x), \rho', \rho_1))$ and $m'_o = (x \Rightarrow \mathbf{coerce}((e'\ x), \rho', \rho_2));$

$\mathbf{coerce}(e', \langle \mathtt{elist} \rightarrow \rho_1, \mathtt{olist} \rightarrow \rho_2 \rangle, \langle \mathtt{elist} \rightarrow \rho'_1, \mathtt{olist} \rightarrow \rho'_2 \rangle) = \underline{\mathtt{fix}}\ d'\ \underline{\mathtt{in}}\ (\underline{\mathtt{fn3}}\ (f_u, f_e, f_o))$
where f_u, f_e, f_o are new program variables
and $d' = (f_u = m'_u)\ \underline{\mathtt{and}}\ (f_e = m'_e)\ \underline{\mathtt{and}}\ (f_o = m'_o)$ and $m'_u = (\mathbf{combine}(f_e, \rho'_1, f_o, \rho'_2))$
and $m'_e = (x \Rightarrow \mathbf{coerce}(((\mathtt{efetch}(e'))\ x), \rho_1, \rho'_1))$ and $m'_o = (x \Rightarrow \mathbf{coerce}(((\mathtt{ofetch}(e'))\ x), \rho_2, \rho'_2))$

$\mathbf{applyfun}(e'_1, \rho_1, e'_2, \rho_2) = (e'_1\ (\mathbf{coerce}(e'_2, \rho_2, \tau_2)), \mathbf{apprfty}(\rho_1, \rho_2))$ if $\rho_1 = \tau_2 \rightarrow \rho$;
$\mathbf{applyfun}(e'_1, \rho_1, e'_2, \rho_2) = ((\mathit{fetch}(e'_1))\ e'_2,\ \mathbf{apprfty}(\rho_1, \rho_2))$ if $\rho_1 = \langle \mathtt{elist} \rightarrow \rho'_1, \mathtt{olist} \rightarrow \rho'_2 \rangle$
here *fetch* is respectively `efetch`, `ofetch`, or `ufetch` if ρ_2 is `elist`, `olist`, or others;
Otherwise, $\mathbf{applyfun}(e'_1, \rho_1, e'_2, \rho_2) = (e'_1 e'_2,\ \mathbf{apprfty}(\rho_1, \rho_2))$.

Figure 6: Definitions of "**combine**", "**coerce**", and "**applyfun**"

get expressions e'_1 and e'_2, and two refinement types ρ_1 and ρ_2, the operation $\mathbf{combine}(e'_1, \rho_1, e'_2, \rho_2)$, constructs a dispatch TGT function that calls e'_1 or e'_2 respectively depending on the length parity of its argument.

It is occasionally necessary to introduce code to coerce the result of a term from one representation to another. Given a target expression e', two refinement types ρ_1 and ρ_2 such that $\rho_1 \leq \rho_2$, then the operation $\mathbf{coerce}(e', \rho_1, \rho_2)$, which returns a new target expression, is defined in Figure 6. Notice that **combine** and **coerce** are mutually recursive.

The **applyfun** operation inserts appropriate coercions for function applications. Given two TGT expressions e'_1 and e'_2, and two refinement types ρ_1 and ρ_2, the operation $\mathbf{applyfun}(e'_1, \rho_1, e'_2, \rho_2)$, which returns a TGT expression and a refinement type, is also defined in Figure 6.