A Functional Representation of Data Structures with a Hole

Yasuhiko Minamide
Research Institute for Mathematical Sciences
Kyoto University
Kyoto 606-01, JAPAN
nan@kurims.kyoto-u.ac.jp

Abstract

Data structures with a hole, in other words data structures with an uninitialized field, are useful to write efficient programs: they enable us to construct functional data structures flexibly and write functions such as append and map as tail recursive functions. In this paper we present an approach to introducing data structures with a hole into call-by-value functional programming languages like ML. Data structures with a hole are formalized as a new form of λ-abstraction called hole abstraction. The novel features of hole abstraction are that expressions inside hole abstraction are evaluated and application is implemented by destructive update of a hole. We present a simply typed call-by-value λ-calculus extended with hole abstractions. Then we show a compilation method of hole abstraction and prove correctness of the compilation.

1 Introduction

Functional data types such as list are distinguished from imperative data types such as ref and array in ML, and no destructive operation is provided for functional data types. This distinction ensures that there is no side effect for functional data types in ML and makes understanding of ML programs easier. However, the way functional data structures are built is very restrictive: they can be built only in the bottom-up manner. This limitation often prevents us from writing efficient programs.

In order to construct functional data structures more flexibly we will introduce data structures with a hole into call-by-value functional languages like ML. By data structures with a hole we mean data structures with an uninitialized field such as a cons cell whose tail is not yet given. Data structures with a hole enable us to build functional data structures more flexibly: functional data structures can be built in the top-down manner. For example, a list can be built from the head to the tail.

Theoretically, data structures with a hole can be represented by usual λ-abstraction: for example, $\lambda x.\mathtt{cons}(1, x)$ can be considered as a cons cell whose tail is a hole. However, this does not have the intended operational behaviour in call-by-value functional programming languages. It is because an expression inside λ-abstraction is not evaluated and thus the data structure is not constructed at the time of the evaluation of abstraction, but at the time of the evaluation of application.

Thus in order to represent data structures with a hole we introduce a new form of λ-abstraction $\hat{\lambda}x.M$ called *hole abstraction*. A hole abstraction is evaluated to a pair of a data structure and the pointer to its hole at runtime. The application of a hole abstraction fills its hole destructively. In order to create a data structure at the time of the evaluation of hole abstraction, an expression inside $\hat{\lambda}$-abstraction is evaluated. This implementation of hole abstraction imposes several restrictions on the usage of hole abstraction.

In order to formalize the idea of hole abstractions, we propose a simply typed call-by-value λ-calculus extended with hole abstractions. The type system of the calculus is designed so that the implementation of hole abstraction and application introduces no side effect and preserves desirable properties of the simply typed call-by-value λ-calculus. We hide side effects by restricting the use of hole abstraction to the single-threaded manner [6]. The restriction is imposed by considering the types of hole abstractions as linear types [21, 22, 16]. Then we give the operational semantics of the calculus and prove the soundness of the type system.

By using hole abstractions we have more flexibility on construction of functional data structures than in the conventional call-by-value functional languages. This enables us to apply some programming techniques used for imperative data types to functional data types. The most significant application is to implement functions such as append and map as tail recursive functions. Furthermore, this tail recursive implementation can be considered as continuation-passing style transformation of the standard implementation of these functions: continuations for recursive calls are represented by hole abstractions.

The next step is to establish a compilation method of hole abstractions. We propose a compilation method which can be incorporated into standard compilers with only minimal changes. It is based on a transformation that converts a general hole abstraction into a composition of primitive hole abstractions. After this transformation we need only the evaluation of primitive hole abstractions and do not require the evaluation of an expression inside hole abstraction. Thus programs can be compiled by a standard compilation method after the transformation. For this transformation, we prove the correctness by the method of logical relations.

We incorporated some ideas of hole abstractions into an experimental ML compiler developed by the author. The results of preliminary benchmarks show about 15 - 40% improvement in execution time.

The rest of this paper is organized as follows. In Section 2 we introduce hole abstraction informally and show how to use hole abstraction to write efficient programs, especially tail recursive functions. In Section 3 we present a simply typed call-by-value λ-calculus extended with hole abstractions and prove the soundness

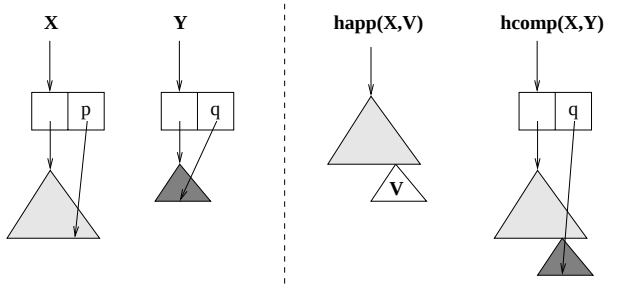

Figure 1: Application and Composition of Hole Abstractions

of the calculus. Then we show a compilation method of hole abstractions in Section 4. In Section 5 we discuss implementation and show how much improvement we can obtain by using hole abstractions. We discuss related work in Section 6, and suggest directions for future work in Section 7.

2 Data Structures with a Hole

In the λ-calculus data structures with a hole can be represented by λ-abstraction. For example, $\lambda x.\texttt{cons}(1, x)$ can be considered as a cons cell whose tail is a hole. In this representation, the operation for filling the hole can be done by application: for example, $(\lambda x.\texttt{cons}(1, x))$ `nil` fills the hole with `nil`.

However, this does not have the intended operational behaviour in call-by-value functional programming languages. It is because an expression inside λ-abstraction is not evaluated. Thus a data structure with a hole is not constructed at the time of the evaluation of λ-abstraction, but at the time of the evaluation of application. Furthermore, it is not a desirable representation for efficiency because creation and application of a closure are not inexpensive operations.

Thus in order to represent data structures with a hole we introduce a new form of λ-abstraction $\hat{\lambda}x.M$ called *hole abstraction*. We also sometimes consider hole abstractions as a class of functions and say *hole functions*. For example, the previous example is represented by $\hat{\lambda}x.\texttt{cons}(1, x)$. For hole abstractions, we give type (τ_1, τ_2) *hfun* where τ_1 is the type of a hole and τ_2 is the type of a resulting data structure.

For efficient implementation of hole abstractions, at run time hole abstractions are represented by a pair of pointers to a data structure and its hole as shown in Figure 1: X and Y are hole abstractions where data structures are represented by triangles, and p and q are the pointers to the holes.

The primitives for hole abstractions are illustrated in Figure 1. The primitive `happ` destructively fills the hole and corresponds to function applications as we saw before. The primitive `hcomp` fills the hole of the first argument with another data structure with a hole and corresponds to function compositions of the λ-calculus. It is because in the λ-calculus the composition of $\hat{\lambda}x.M$ and $\hat{\lambda}y.N$ can be reduced to $\hat{\lambda}y.M[N/x]$ which represents the composed data structure with a hole. For example, $\texttt{hcomp}(\hat{\lambda}x.\texttt{cons}(1, x), \hat{\lambda}y.\texttt{cons}(2, y))$ should be evaluated to $\hat{\lambda}y.\texttt{cons}(1, x)[\texttt{cons}(2, y)/x] \equiv \hat{\lambda}y.\texttt{cons}(1, \texttt{cons}(2, y))$.

In this representation, the identity hole abstraction, $\hat{\lambda}x.x$, cannot be treated because its body does not construct any data structure, but it is just a hole. However, the identity hole abstraction is useful to write programs as we see later in examples. Thus we represent the identity hole abstraction, $\hat{\lambda}x.x$, by a pair of special values which can be distinguished from other hole abstractions. [1]

This implementation of hole abstractions and primitives for them imposes several restrictions on the usage of hole abstractions.

First, the destructive implementation of `happ` and `hcomp` must be hidden so that no side effect can be seen. This is solved by restricting operations on hole abstractions to the single-threaded manner as proposed for introducing imperative operations in pure functional languages [6, 21]. We impose single-threadedness by considering (τ_1, τ_2) *hfun* as linear types. This makes efficient implementation through destructive update possible without introducing side effects.

Secondly, an expression inside $\hat{\lambda}$-abstraction must be restricted so that the expression can be evaluated to a data structure without using the value of a variable introduced by $\hat{\lambda}$-abstraction. If we impose no restriction on an expression inside $\hat{\lambda}$-abstraction, we cannot obtain a data structure as a result of the evaluation of the expression. Let us consider the following program:

$$\hat{\lambda}x.\texttt{case}\ x\ \texttt{of nil} \Rightarrow 0 \mid\ \texttt{cons}(y, z) \Rightarrow 1$$

In this program, x is used in the test of `case`-expression. However, the value of x is not yet determined at the time of the evaluation. Thus we cannot evaluate this expression to a data structure. We also prohibit application and λ-abstraction containing free variables introduced by hole abstractions. For example, expressions $\hat{\lambda}x.yx$ and $\hat{\lambda}x.\lambda y.x$ are not permitted. This restriction is necessary for our implementation strategy.

Thirdly, hole abstractions must contain exactly one hole. It is because a hole abstraction is implemented as a pair of pointers to a data structure and a single hole. Thus a data structure with no hole or multiple holes cannot be represented. For example, the following programs are not permitted: $\hat{\lambda}x.\texttt{nil}$ and $\hat{\lambda}x.\texttt{cons}(x, \texttt{cons}(x, \texttt{nil}))$.

2.1 Using Hole Abstractions

In this section we will explain how to use hole abstractions to improve programs. For example programs we use the syntax of Standard ML and `hfn x => M` instead of $\hat{\lambda}x.M$.

By using hole abstractions we can build functional data structures more flexibly than in the conventional call-by-value functional languages. Functional data structures can be built in the top-down manner. For example, the following program creates a list containing 1 and 2 from the head.

```
let val y1 = hfn x => 1::x
    val y2 = hcomp (y1, hfn x => 2::x)
in
    happ (y2, nil)
end
```

This example starts to create the list from a cons cell where its head is 1 and its tail is a hole. Another way is to create a list from the identity hole abstraction as follows:

```
let val y0 = hfn x => x
    val y1 = hcomp (y0, hfn x => 1::x)
    val y2 = hcomp (y1, hfn x => 2::x)
in
    happ (y2, nil)
end
```

[1] By including the identity hole abstraction, the implementation of `happ` and `hcomp` must check whether arguments are the identity hole abstraction.

```
fun append ([],ys) = ys
  | append (x::xs,ys) = x :: append (xs, ys)

fun hfun_append (xs, ys) =
    let fun append_rec ([], k) = happ (k, ys)
          | append_rec (x::xs, k) = append_rec (xs, hcomp (k, hfn z => x::z))
    in
        append_rec (xs, hfn x => x)
    end

fun flatten [] = []
  | flatten (x::xs) = append (x, flatten xs)

fun append' (k, []) = k
  | append' (k, x::xs) = append' (hcomp (k, hfn y => x::y), xs)

fun hfun_flatten xs =
    let fun flatten_rec (k, []) = happ (k, [])
          | flatten_rec (k, x::xs) = flatten_rec (append' (k, x), xs)
    in
        flatten_rec (hfn x => x,  xs)
    end
```

Figure 2: Using Hole Abstractions for Lists

This kind of the usage of the identity hole abstraction is often useful to write programs as we see later.

By using data structures with a hole we can write functions which usually need to construct a data structure by using the return value of a recursive call as tail recursive functions.

Let us consider the function `append` in Figure 2. This function is not tail recursive because consing must be performed after the recursive call of `append`. By using hole abstractions the same function can be implemented as the tail recursive function `hfun_append` in Figure 2. For the argument `k` of `append_rec`, we use hole abstractions representing lists whose tail is a hole. The expression `hcomp (k, hfn z => x::z)` updates the tail of `k` by the new cons cell consisting of `x` and a hole. That is to say, it adds `x` to the right of `k`. Finally, the result is obtained by updating the tail of `k` by `ys`, i.e., `happ(k,ys)`. The identity hole abstraction is used for the initial argument. To sum up, `hfun_append` creates an appended list from the head. In our measurement this transformation reduces execution time about 40% as we see later in Section 5.

Another view of this transformation is continuation passing transformation [17]. If we think `k` as a continuation and replace hole functions by usual functions, we obtain `append` in continuation passing style. Thus it can be considered that continuations are represented by hole functions. However, if we use usual functions for continuations, the performance of the function is usually not improved. It is because the function for the continuation grows through recursive calls as stack grows for the original recursive version.

There are many functions producing a list which can be transformed to tail recursive functions. For example, there are 12 functions producing lists in the structure for lists in the proposal of Standard ML Basis Library. Among them, the following 6 functions can be converted to tail recursive functions in this way: `@`, `take`, `map`, `mapPartial`, `filter`, `tabulate`.

Furthermore, the transformation can be implemented as optimization that transforms a class of functions into tail recursive functions automatically. We can check whether the transformation can be applicable syntactically by checking whether the continuations for recursive calls can be represented by hole abstractions. This is a significant advantage of our transformation over the conventional optimization which transforms a function into tail recursive one [2, 7]. The conventional transformation usually needs the knowledge of properties such as associativity of functions for the transformation.

The previous improvement of programs can be obtained by a compiler optimization without explicitly introducing hole abstractions into source languages. However, there are other kinds of usages of hole abstractions. The function `flatten` in Figure 2 flattens a list of lists to a list by using `append`. The function `flatten` is not tail recursive because `append` must be performed after the recursive call of `flatten`. For better implementation of `flatten`, we first prepare the function `append'` which has type `('a list, 'a list) hfun -> ('a list, 'a list) hfun` and appends a list to a list whose tail is a hole. Then we can implement `flatten` as the tail recursive function `hfun_flatten`. This function `hfun_flatten` can be compiled to efficient code with nested loops by inlining the definition of `append'`. However, this definition of `hfun_flatten` can be obtained only by using associativity of `append` and the following fact: if `happ(k, []) = l` then `happ(append'(k, x),[]) = append (l,x)`. Without knowing these properties it is not possible to obtain `hfun_flatten` by the compiler optimization.

Hole abstractions are useful for other data types than lists. In Figure 3, hole abstractions are used for the type `tree` that is the type for binary trees whose nodes are associated to integers. The function `binsert` inserts an integer into a tree and is not tail recursive. As `append`, the function can be transformed to the tail recursive function `hfun_binsert` below by using hole abstractions.

However, for data types like trees there are many functions that cannot be transformed to tail recursive form. For such functions, although we cannot obtain tail recursive functions, we can often improve functions by converting the last recursive call to a tail recursive call. For example, `addone` which adds 1 to the integer as-

```
datatype tree = Lf
              | Br of int *  tree * tree

fun binsert (Lf, y) = Br (y, Lf, Lf)
  | binsert (Br (x, t1, t2), y) =
    if y < x then Br (x, binsert  (t1, y), t2)
    else if x < y then Br (x, t1, binsert  (t2, y)) else raise Bsearch

fun hfun_binsert (t, y) =
    let fun binsert' (Lf, y, k) = happ (k, Br (y, Lf, Lf))
          | binsert' (Br (x, t1, t2), y, k) =
        if y < x then binsert'  (t1, y, hcomp (k, hfn t => Br (x, t, t2)))
        else
            if x < y then binsert'  (t2, y, hcomp (k, hfn t => Br (x, t1, t)))
            else raise Bsearch
    in
        binsert' (t, y, hfn t => t)
    end

fun addone Lf  = Lf
  | addone (Br (x, t1, t2)) = Br (x + 1, addone t1, addone t2)

fun hfun_addone Lf  = Lf
  | hfun_addone (Br (x, t1, t2)) =
    let fun addone' (Lf, k) = happ (k, Lf)
          | addone' (Br (x, t1, t2), k)  =
        addone' (t2, hcomp (k, hfn t2 => Br (x + 1, hfun_addone t1, t2)))
    in
        addone' (t2, hfn t2 => Br (x + 1, hfun_addone t1, t2))
    end
```

Figure 3: Using Hole Abstractions for Trees

sociated to each node includes two recursive calls as shown in Figure 3. Thus it cannot be converted to a tail recursive function by our method. However, the function can be converted to `hfun_addone` that requires one recursive call of `hfun_addone` and another tail recursive call of `addone'`. It should be remarked that the expressions `x + 1` and `hfun_addone t1` are evaluated at the time of evaluation of `hfn t2 => ...`, because an expression inside `hfn` is evaluated. It is possible because they do not contain the variable `t2` introduced by hole abstraction.

3 Calculus

In this section we present a simply typed call-by-value λ-calculus extended with hole abstractions. We design the type system of the calculus so that the restrictions discussed in the previous section are imposed. Then we will show some properties of the calculus including the soundness of the type system. The syntax of the calculus is defined as follows:

$$\begin{array}{lrcl} \textit{types} & \tau & ::= & b \mid \tau_1 \to \tau_2 \mid \tau\ \textit{list} \mid (\tau_1, \tau_2)\ \textit{hfun} \\ \textit{exp's} & M & ::= & x \mid c \mid M_1\, M_2 \mid \lambda x:\tau.M \mid \\ & & & \texttt{nil} \mid \texttt{cons}(M_1, M_2) \mid \\ & & & \hat{\lambda} x:\tau.M \mid \texttt{happ}(M_1, M_2) \mid \\ & & & \texttt{case}\ M_1\ \texttt{of}\ \texttt{cons}(x_1, x_2) \Rightarrow M_2 \mid \texttt{nil} \Rightarrow M_3 \end{array}$$

We include τ *list* as a type constructor, and `nil` and $\texttt{cons}(M_1, M_2)$ as expressions for illustration of usage of the calculus and its compilation. The hole composition $\texttt{hcomp}(M_1, M_2)$ is not included in the calculus because it can be defined by hole abstractions and applications as we will see later.

We consider types containing $hfun$ as linear types, though $\tau_1 \to \tau_2$ is always considered a nonlinear type. Thus linear types are defined as follows:

$$\sigma ::= (\tau_1, \tau_2)\ \textit{hfun} \mid \sigma\ \textit{list}$$

The treatment of linear types is based on steadfast types of Wadler [21, 22]. The type system is designed so that a variable with a linear type contains the sole pointer [23] and thus we can perform destructive operations on values of linear types. However, since linearity is used only for destructive update, values of linear types are permitted to be abandoned implicitly. In this sense they are considered as affine types of Jacobs [8] or unique types of Clean [16].

For the type system of the calculus we use two kinds of contexts:

$$\begin{array}{lrcl} \textit{type assignements} & \Gamma & ::= & x_1{:}\tau_1, \ldots, x_n{:}\tau_n \\ \textit{hole contexts} & H & ::= & \emptyset \mid x{:}\tau \end{array}$$

Type assignments are used for variables introduced by usual λ-abstraction and hole contexts are used for variables introduced by $\hat{\lambda}$-abstraction. Here we should remark that hole contexts contain at most one binding. By this restriction we prohibit data structures with multiple holes represented by nested hole abstractions. For example, $\hat{\lambda} x:\tau.\hat{\lambda} y:\tau\ \textit{list}.\texttt{cons}(x, y)$ is prohibited. This restriction

$$\text{(hole)}\quad \Gamma; x{:}\tau \vdash x : \tau \qquad \text{(var)}\quad \frac{x{:}\tau \in \Gamma}{\Gamma; \emptyset \vdash x : \tau}$$

$$\text{(hfn)}\quad \frac{\Gamma; x{:}\tau_1 \vdash M : \tau_2}{\Gamma; \emptyset \vdash \hat{\lambda} x{:}\tau_1.M : (\tau_1, \tau_2)\ \mathit{hfun}} \qquad \text{(happ)}\quad \frac{\Gamma_1; \emptyset \vdash M_1 : (\tau_1, \tau_2)\ \mathit{hfun} \quad \Gamma_2; H \vdash M_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2; H \vdash \mathtt{happ}(M_1, M_2) : \tau_2}$$

$$\text{(nil)}\quad \Gamma; \emptyset \vdash \mathtt{nil} : \tau\ \mathit{list} \qquad \text{(cons)}\quad \frac{\Gamma_1; H_1 \vdash M_1 : \tau \quad \Gamma_2; H_2 \vdash M_2 : \tau\ \mathit{list}}{\Gamma_1 \uplus \Gamma_2; H_1 \oplus H_2 \vdash \mathtt{cons}(M_1, M_2) : \tau\ \mathit{list}}$$

$$\text{(abs)}\quad \frac{\Gamma|_N \uplus \{x : \tau_1\}; \emptyset \vdash M : \tau_2}{\Gamma; \emptyset \vdash \lambda x : \tau_1.M : \tau_1 \to \tau_2} \qquad \text{(app)}\quad \frac{\Gamma_1; \emptyset \vdash M_1 : \tau_1 \to \tau_2 \quad \Gamma_2; \emptyset \vdash M_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2; \emptyset \vdash M_1 M_2 : \tau_2}$$

$$\text{(case)}\quad \frac{\Gamma_1; \emptyset \vdash M : \tau'\ \mathit{list} \quad \Gamma_2, x{:}\tau', y{:}\tau'\ \mathit{list}; \emptyset \vdash M_1 : \tau \quad \Gamma_2; \emptyset \vdash M_2 : \tau}{\Gamma_1 \uplus \Gamma_2; \emptyset \vdash \mathtt{case}\ M\ \mathtt{of}\ \mathtt{cons}(x, y) \Rightarrow M_1\ \mathtt{|}\ \mathtt{nil} \Rightarrow M_2 : \tau}$$

Figure 4: Type System

is not essential for the definition of the calculus, but it is required for the simple compilation method presented in the next section.

We say that Γ_1 and Γ_2 are compatible if for $x{:}\tau \in \Gamma_1$ and $x{:}\tau' \in \Gamma_2$, τ and τ' are identical and nonlinear. For compatible Γ_1 and Γ_2, we define $\Gamma_1 \uplus \Gamma_2$ by the union of the two type assignments. For type assignments we define $\Gamma|_N$ that is the restriction of Γ to $x{:}\tau$ for nonlinear τ. We say that H_1 and H_2 are compatible if either H_1 or H_2 is empty. For compatible hole contexts we define $H_1 \oplus H_2$ by the union of H_1 and H_2.

The judgement of the type system has the following form: $\Gamma; H \vdash M : \tau$. The type system of the calculus is shown in Figure 4. We summarize the key features of the type system below.

- Variables introduced by λ-abstractions can be abandoned by the rule (var). However, even for nonlinear types a variable introduced by $\hat{\lambda}$-abstraction is used linearly in a strict sense and cannot be abandoned implicitly. For example, $\hat{\lambda} x{:}b.\mathtt{nil}$ is ill-typed.
- λ-abstractions, applications, and `case`-expressions can be typed only with the empty hole context. However, they can be still used inside $\hat{\lambda}$-abstraction if the evaluation of them is not related to the hole. For example, $\hat{\lambda} x{:}b\mathit{list}.\mathtt{cons}((\lambda x{:}b.x)c, x)$ is well-typed.
- In the rule (happ), M_1 of $\mathtt{happ}(M_1, M_2)$ must be typed with the empty hole context. This restriction is imposed because we cannot evaluate expressions such as $\hat{\lambda} x{:}(b, \tau)\ \mathit{hfun}.\mathtt{happ}(x, c)$.
- In the rule (abs), only the nonlinear part of Γ, $\Gamma|_N$, is used for the typing of the body of a λ-abstraction. It is because $\tau_1 \to \tau_2$ is considered as a nonlinear type.

We define the operational semantics of the calculus in the style of [25] and prove the soundness of the type system. The operational semantics of the calculus is defined in Figure 5. We consider a $\hat{\lambda}$-abstraction as a value only if the body of the abstraction is a value. Evaluation contexts are defined so that evaluation does not occur inside usual λ-abstraction, but inside $\hat{\lambda}$-abstraction. Similar evaluation order is used for the implementation calculus of polymorphic records of Ohori [13] where evaluation occurs within index abstraction to preserve the operational semantics of ML. The definition of reductions is standard and is also shown in Figure 5.

Then evaluation $M \longmapsto M'$ is defined by $E[M] \longmapsto E[M']$ if $M \blacktriangleright M'$. We write $M \downarrow M'$ if $M \longmapsto^* M'$ and M' is in normal form with respect to $\longmapsto$.

The composition $\mathtt{hcomp}(M_1, M_2)$ can be defined in this calculus as $\hat{\lambda} x{:}\tau.\mathtt{happ}(M_1, \mathtt{happ}(M_2, x))$ and the reductions inside $\hat{\lambda}$-abstractions result in the intended simplified hole abstraction as follows:

$$\begin{array}{l}\mathtt{hcomp}(\hat{\lambda} y_1{:}\tau'.V_1, \hat{\lambda} y_2{:}\tau.V_2) \longmapsto^* \\ \hat{\lambda} x{:}\tau.\mathtt{happ}(\hat{\lambda} y_1{:}\tau'.V_1, V_2[x/y_2]) \longmapsto \\ \hat{\lambda} x{:}\tau.V_1[V_2[x/y_2]/y_1]\end{array}$$

The reduction of `hcomp` should be compared to that of the usual function composition:

$$(\lambda x{:}\tau'.M) \circ (\lambda y{:}\tau.N) \longmapsto^* \lambda z{:}\tau.(\lambda x{:}\tau'.M)((\lambda y{:}\tau.N)z)$$

Here the original two lambda abstractions still remain even after reduction.

The following two lemmas show that the evaluation preserves the type of an expression.

Lemma 1 (Substitution) *1. If $\Gamma \uplus x{:}\tau_0; H \vdash M : \tau$ and $\emptyset; \emptyset \vdash V : \tau_0$, then $\Gamma; H \vdash M[V/x] : \tau$.*

2. If $\emptyset; x{:}\tau_0 \vdash V_1 : \tau$ and $\emptyset; H \vdash V_2 : \tau_0$, then $\emptyset; H \vdash V_1[V_2/x] : \tau$.

Lemma 2 (Subject Reduction) *If $\emptyset; H \vdash M{:}\tau$ and $M \longmapsto M'$, then $\emptyset; H \vdash M'{:}\tau$.*

The most important lemma is the following: it claims that the evaluation of a program cannot get stuck [25].

Lemma 3 *If $\emptyset; H \vdash M{:}\tau$, then M is a value or $M \longmapsto M'$ for some M'.*

By combining the previous two lemmas, we obtain the soundness of the type system.

Theorem 1 (Soundness) *If $\emptyset; \emptyset \vdash M{:}\tau$, then M is a value or $M \longmapsto M'$ and $\emptyset; \emptyset \vdash M'{:}\tau$.*

Although the calculus we have considered does not include recursion, we think that it is easy to extend the calculus by adding recursion.

Values:

$$V ::= c \mid x \mid \lambda x{:}\tau.M \mid \texttt{nil} \mid \texttt{cons}(V_1, V_2) \mid \hat{\lambda} x{:}\tau.V$$

Evaluation Contexts:

$$\begin{array}{rcl} E & ::= & [] \mid E\,M \mid V\,E \mid \\ & & \texttt{cons}(E, M) \mid \texttt{cons}(V, E) \mid \texttt{case}\ E\ \texttt{of}\ \texttt{cons}(x, y) \Rightarrow M_1 \mid \texttt{nil} \Rightarrow M_2 \mid \\ & & \hat{\lambda} x{:}\tau.E \mid \texttt{happ}(E, M) \mid \texttt{happ}(V, E) \end{array}$$

Reduction Rules:

$$\begin{array}{rcl} (\lambda x{:}\tau.M)V & \blacktriangleright & M[V/x] \\ \texttt{happ}(\hat{\lambda} x{:}\tau.V_1, V_2) & \blacktriangleright & V_1[V_2/x] \\ \texttt{case cons}(V_1, V_2)\ \texttt{of cons}(x, y) \Rightarrow M_1 \mid \texttt{nil} \Rightarrow M_2 & \blacktriangleright & M_1[V_1/x, V_2/y] \\ \texttt{case nil of cons}(x, y) \Rightarrow M_1 \mid \texttt{nil} \Rightarrow M_2 & \blacktriangleright & M_2 \end{array}$$

Figure 5: Operational Semantics

4 Compilation

In this section we will show a compilation method of hole abstractions. It is based on a transformation that converts a general hole abstraction into a composition of primitive hole abstractions. This transformation is analogous to the translation from the λ-calculus to combinatory logic found in the standard text [1]. After this transformation we need only the evaluation of primitive hole abstractions and do not require the evaluation of expressions inside hole abstractions. Thus programs can be compiled by a standard compilation method after the transformation. We prove the correctness of the transformation by the method of logical relations.

In this section, we consider `hcomp` as a primitive that is implemented as we described in Section 2. The reduction rule for `hcomp` is defined as follows:

$$\texttt{hcomp}(\hat{\lambda} y_1{:}\tau'.V_1, \hat{\lambda} y_2{:}\tau.V_2) \blacktriangleright \hat{\lambda} y_2{:}\tau.V_1[V_2/y_1]$$

Furthermore, we consider three more hole functions as primitives: $\texttt{idhfun}_\tau$, $\texttt{conshead}_\tau(M)$, and $\texttt{constail}_\tau(M)$. They correspond to $\hat{\lambda} x{:}\tau.x$, $\hat{\lambda} x{:}\tau.\texttt{cons}(x, M)$, and $\hat{\lambda} x{:}\tau\ \textit{list}.\texttt{cons}(M, x)$ respectively. The reduction rules for them are defined as follows:

$$\begin{array}{rcl} \texttt{idhfun}_\tau & \blacktriangleright & \hat{\lambda} x{:}\tau.x \\ \texttt{conshead}_\tau(V) & \blacktriangleright & \hat{\lambda} x{:}\tau.\texttt{cons}(x, V) \\ \texttt{constail}_\tau(V) & \blacktriangleright & \hat{\lambda} x{:}\tau\ \textit{list}.\texttt{cons}(V, x) \end{array}$$

The typing rules for the primitives are derived from the definition and shown in Figure 6. We also extend evaluation contexts as follows:

$$\begin{array}{rcl} E & ::= & \ldots \mid \texttt{hcomp}(E, M) \mid \texttt{hcomp}(V, E) \mid \\ & & \texttt{conshead}_\tau(E) \mid \texttt{constail}_\tau(E) \end{array}$$

The compilation is defined as a deductive system with judgements of the following form:

$$\Gamma; H \vdash M : \tau \rightsquigarrow M'.$$

The rules of the deductive system are shown in Figure 7. The rules can be considered as refinements of the typing rules. Thus it is clear that if $\Gamma; H \vdash M : \tau$, then $\Gamma; H \vdash M : \tau \rightsquigarrow M'$ for some M'. Conversely, if $\Gamma; H \vdash M : \tau \rightsquigarrow M'$, then $\Gamma; H \vdash M : \tau$.

There are three rules for `cons`: (cons) is used if the hole context is empty, (conshead) and (constail) are used if the hole context is not empty. There are two rules for `happ` depending on whether the hole context is empty or not.

For example, $\hat{\lambda} x{:}\textit{int list}.\texttt{cons}((\lambda y{:}\textit{int}.y)1, x)$ is translated to the following expression.

$$\texttt{hcomp}(\texttt{constail}_{\textit{int}}((\lambda y{:}\textit{int}.y)1), \texttt{idhfun}_{\textit{int list}})$$

This expression has type $(\textit{int list}, \textit{int list})$ *hfun* as we expect. Furthermore, it is evaluated as follows:

$$\begin{array}{rl} & \texttt{hcomp}(\texttt{constail}_{\textit{int}}((\lambda y{:}\textit{int}.y)1), \texttt{idhfun}_{\textit{int list}}) \\ \longmapsto & \texttt{hcomp}(\texttt{constail}_{\textit{int}}(1), \texttt{idhfun}_{\textit{int list}}) \\ \longmapsto & \texttt{hcomp}(\hat{\lambda} x{:}\textit{int list}.\texttt{cons}(1, x), \texttt{idhfun}_{\textit{int list}}) \\ \longmapsto & \texttt{hcomp}(\hat{\lambda} x{:}\textit{int list}.\texttt{cons}(1, x), \hat{\lambda} x{:}\textit{int list}.x) \\ \longmapsto & \hat{\lambda} x{:}\textit{int list}.\texttt{cons}(1, x) \end{array}$$

This evaluation does not require reductions inside $\hat{\lambda}$-abstractions.

Let us check that the derived definition of hole composition in the previous section is actually compiled to the primitive hole composition by this translation. The derived definition of hole composition of M_1 and M_2 is $\hat{\lambda} x{:}\tau.\texttt{happ}(M_1, \texttt{happ}(M_2, x))$. This expression is translated to $\texttt{hcomp}(M_1', \texttt{hcomp}(M_2', \texttt{idhfun}_\tau))$. Then it is easily shown that this expression is equivalent to $\texttt{hcomp}(M_1', M_2')$.

The translation rules often introduce redundant identity hole abstractions. For example, $\hat{\lambda} x.\texttt{cons}(1, x)$ is translated to $\texttt{hcomp}(\texttt{constail}(1), \texttt{idhfun}_{\textit{int list}})$ and it is clearly equivalent to $\texttt{constail}(1)$. In general it is easily shown that $\texttt{hcomp}(M, \texttt{idhfun}_\tau)$ is equivalent to M. By considering that we obtain the following derived rules. They can be added for removing redundant identity hole abstractions.

$$\frac{\Gamma; \emptyset \vdash M_2 : \tau\ \textit{list} \rightsquigarrow M_2'}{\Gamma; x : \tau \vdash \texttt{cons}(x, M_2) : \tau\ \textit{list} \rightsquigarrow \texttt{conshead}(M_2')}$$

$$\frac{\Gamma; \emptyset \vdash M_1 : \tau \rightsquigarrow M_1'}{\Gamma; x : \tau\ \textit{list} \vdash \texttt{cons}(M_1, x) : \tau\ \textit{list} \rightsquigarrow \texttt{constail}(M_1')}$$

$$\frac{\Gamma; \emptyset \vdash M_1 : (\tau_1, \tau_2)\ \textit{hfun} \rightsquigarrow M_1'}{\Gamma; x : \tau_1 \vdash \texttt{happ}(M_1, x) : \tau_2 \rightsquigarrow M_1'}$$

$$\Gamma;\emptyset \vdash \texttt{idhfun}_\tau : (\tau,\tau)\ \textit{hfun} \qquad \frac{\Gamma;\emptyset \vdash M_1 : (\tau_1,\tau_2)\ \textit{hfun} \quad \Gamma;\emptyset \vdash M_2 : (\tau_3,\tau_1)\ \textit{hfun}}{\Gamma;\emptyset \vdash \texttt{hcomp}(M_1,M_2) : (\tau_3,\tau_2)\ \textit{hfun}}$$

$$\frac{\Gamma;\emptyset \vdash M : \tau\ \textit{list}}{\Gamma;\emptyset \vdash \texttt{conshead}_\tau(M) : (\tau,\tau\ \textit{list})\ \textit{hfun}} \qquad \frac{\Gamma;\emptyset \vdash M : \tau}{\Gamma;\emptyset \vdash \texttt{constail}_\tau(M) : (\tau\ \textit{list},\tau\ \textit{list})\ \textit{hfun}}$$

Figure 6: Typing Rules for Primitives

(var) $\dfrac{x{:}\tau \in \Gamma}{\Gamma;\emptyset \vdash x : \tau \rightsquigarrow x}$ (nil) $\Gamma;\emptyset \vdash \texttt{nil} : \tau\ \textit{list} \rightsquigarrow \texttt{nil}$

(hfn) $\dfrac{\Gamma; x : \tau_1 \vdash M : \tau_2 \rightsquigarrow M'}{\Gamma;\emptyset \vdash \hat{\lambda} x : \tau_1.M : (\tau_1,\tau_2)\ \textit{hfun} \rightsquigarrow M'}$ (hole) $\Gamma; x : \tau \vdash x : \tau \rightsquigarrow \texttt{idhfun}_\tau$

(conshead) $$\frac{\Gamma_1; x : \tau \vdash M_1 : \tau' \rightsquigarrow M_1' \quad \Gamma_2;\emptyset \vdash M_2 : \tau'\ \textit{list} \rightsquigarrow M_2'}{\Gamma_1 \uplus \Gamma_2; x : \tau \vdash \texttt{cons}(M_1,M_2) : \tau'\ \textit{list} \rightsquigarrow \texttt{hcomp}(\texttt{conshead}_{\tau'}(M_2'),M_1')}$$

(constail) $$\frac{\Gamma_1;\emptyset \vdash M_1 : \tau' \rightsquigarrow M_1' \quad \Gamma_2; x : \tau \vdash M_2 : \tau'\ \textit{list} \rightsquigarrow M_2'}{\Gamma_1 \uplus \Gamma_2; x : \tau \vdash \texttt{cons}(M_1,M_2) : \tau'\ \textit{list} \rightsquigarrow \texttt{hcomp}(\texttt{constail}_{\tau'}(M_1'),M_2')}$$

(cons) $$\frac{\Gamma_1;\emptyset \vdash M_1 : \tau' \rightsquigarrow M_1' \quad \Gamma_2;\emptyset \vdash M_2 : \tau'\ \textit{list} \rightsquigarrow M_2'}{\Gamma_1 \uplus \Gamma_2;\emptyset \vdash \texttt{cons}(M_1,M_2) : \tau'\ \textit{list} \rightsquigarrow \texttt{cons}(M_1',M_2')}$$

(happ) $$\frac{\Gamma_1;\emptyset \vdash M_1 : (\tau_1,\tau_2)\ \textit{hfun} \rightsquigarrow M_1' \quad \Gamma_2;\emptyset \vdash M_2 : \tau_1 \rightsquigarrow M_2'}{\Gamma_1 \uplus \Gamma_2;\emptyset \vdash \texttt{happ}(M_1,M_2) : \tau_2 \rightsquigarrow \texttt{happ}(M_1',M_2')}$$

(hcomp) $$\frac{\Gamma_1;\emptyset \vdash M_1 : (\tau_1,\tau_2)\ \textit{hfun} \rightsquigarrow M_1' \quad \Gamma_2; x{:}\tau \vdash M_2 : \tau_1 \rightsquigarrow M_2'}{\Gamma_1 \uplus \Gamma_2; x{:}\tau \vdash \texttt{happ}(M_1,M_2) : \tau_2 \rightsquigarrow \texttt{hcomp}(M_1',M_2')}$$

(abs) $$\frac{\Gamma|_N \uplus \{x : \tau_1\};\emptyset \vdash M : \tau_2 \rightsquigarrow M'}{\Gamma;\emptyset \vdash \lambda x : \tau_1.M : \tau_1 \to \tau_2 \rightsquigarrow \lambda x : \tau_1.M'}$$

(app) $$\frac{\Gamma_1;\emptyset \vdash M_1 : \tau_1 \to \tau_2 \rightsquigarrow M_1' \quad \Gamma_2;\emptyset \vdash M_2 : \tau_1 \rightsquigarrow M_2'}{\Gamma_1 \uplus \Gamma_2;\emptyset \vdash M_1 M_2 : \tau_2 \rightsquigarrow M_1' M_2'}$$

(case) $$\frac{\Gamma_1;\emptyset \vdash M : \tau'\ \textit{list} \rightsquigarrow M' \quad \Gamma_2, x{:}\tau', y{:}\tau'\ \textit{list};\emptyset \vdash M_1 : \tau \rightsquigarrow M_1' \quad \Gamma_2;\emptyset \vdash M_2 : \tau \rightsquigarrow M_2'}{\begin{array}{l}\Gamma_1 \uplus \Gamma_2;\emptyset \vdash \texttt{case}\ M\ \texttt{of}\ \texttt{cons}(x,y) \Rightarrow M_1\ \texttt{|}\ \texttt{nil} \Rightarrow M_2 : \tau \\ \qquad\qquad \rightsquigarrow \texttt{case}\ M'\ \texttt{of}\ \texttt{cons}(x,y) \Rightarrow M_1'\ \texttt{|}\ \texttt{nil} \Rightarrow M_2'\end{array}}$$

Figure 7: Rules of Compilation

$M \sim_\tau M'$ $\quad$ $M \downarrow V$ and $M' \downarrow V'$ and $V \approx_\tau V'$

$c \approx_b c$

$\texttt{nil} \approx_{\tau\ \textit{list}} \texttt{nil}$

$\texttt{cons}(V_1,V_2) \approx_{\tau\ \textit{list}} \texttt{cons}(V_1',V_2')$ $\quad$ $V_1 \approx_\tau V_1'$ and $V_2 \approx_{\tau\ \textit{list}} V_2'$

$V \approx_{\tau_1 \to \tau_2} V'$ $\quad$ for $V_1 \approx_{\tau_1} V_1'$ it holds $VV_1 \sim_{\tau_2} V'V_1'$

$V \approx_{(\tau_1,\tau_2)\ \textit{hfun}} V'$ $\quad$ for $V_1 \approx_{\tau_1} V_1'$ it holds $\texttt{happ}(V,V_1) \sim_{\tau_2} \texttt{happ}(V',V_1')$

Figure 8: Logical Relations

For example, by using these rules the derived definition of hole composition $\hat{\lambda}x{:}\tau.\mathtt{happ}(M_1, \mathtt{happ}(M_2, x))$ can be compiled to $\mathtt{hcomp}(M_1', M_2')$ directly.

We define type τ^H as follows: $\tau^{\emptyset} \equiv \tau$ and $\tau^{x:\tau'} \equiv (\tau', \tau)$ *hfun*. Then the following lemma is proved easily by induction of the derivation of compilation.

Lemma 4 (Type Correctness) *If* $\Gamma; H \vdash M : \tau \rightsquigarrow M'$, *then* $\Gamma; \emptyset \vdash M' : \tau^H$.

This lemma implies that the transformation preserves the type of a program.

To prove the operational correctness of the transformation, we use the method of logical relations as many studies on compilation [14, 12]. First, the type-indexed relations $\sim_\tau$ between closed expressions of type τ and relations $\approx_\tau$ between closed values of type τ are defined in Figure 8. We define the relations on list types, $\approx_{\tau\ list}$, by the smallest relations satisfying the condition in Figure 8. Then the relations are well-defined by induction on the structure of types. Informally, $M \sim_\tau M'$ means that M and M' have the same operational behaviour. The relations $\approx_\tau$ are extended to the relations $\approx_\Gamma$ between substitutions. The following lemma states the key properties of hole functions.

Lemma 5 *1. If* $\emptyset; x{:}\tau_3 \vdash \mathtt{happ}(V_1, V_2) : \tau$ *and* $\emptyset; \emptyset \vdash V : \tau_3$, *then* $\mathtt{happ}(\hat{\lambda}x{:}\tau_3.\mathtt{happ}(V_1, V_2), V) \sim_\tau \mathtt{happ}(V_1, \mathtt{happ}(\hat{\lambda}x{:}\tau_3.V_2, V))$.

2. If $\emptyset; \emptyset \vdash \mathtt{hcomp}(V_1, V_2) : (\tau_3, \tau_2)$ *hfun and* $\emptyset; \emptyset \vdash V : \tau_3$, *then* $\mathtt{happ}(\mathtt{hcomp}(V_1, V_2), V) \sim_{\tau_2} \mathtt{happ}(V_1, \mathtt{happ}(V_2, V))$.

3. If $\emptyset; x{:}\tau' \vdash \mathtt{cons}(V_1, V_2) : \tau\ list$ *and* $x \in FV(V_1)$ *and* $\emptyset; \emptyset \vdash V : \tau'$, *then* $\mathtt{happ}(\hat{\lambda}x{:}\tau'.\mathtt{cons}(V_1, V_2), V) \sim_{\tau\ list} \mathtt{cons}(\mathtt{happ}(\hat{\lambda}x{:}\tau'.V_1, V), V_2)$.

4. If $\emptyset; x{:}\tau' \vdash \mathtt{cons}(V_1, V_2) : \tau\ list$ *and* $x \in FV(V_2)$ *and* $\emptyset; \emptyset \vdash V : \tau'$, *then* $\mathtt{happ}(\hat{\lambda}x{:}\tau'.\mathtt{cons}(V_1, V_2), V) \sim_{\tau\ list} \mathtt{cons}(V_1, \mathtt{happ}(\hat{\lambda}x{:}\tau'.V_2, V))$.

Now we will show that an expression of type τ and its translation are related by $\sim_\tau$. We use the previous lemma for the proofs of translations under non-empty hole contexts.

Lemma 6 *1. If* $\Gamma; \emptyset \vdash M : \tau \rightsquigarrow M'$ *and* $\gamma \approx_\Gamma \gamma'$, *then* $\gamma(M) \sim_\tau \gamma'(M')$.

2. If $\Gamma; x : \tau' \vdash M : \tau \rightsquigarrow M'$ *and* $\gamma \approx_\Gamma \gamma'$, *then* $\gamma(\hat{\lambda}x{:}\tau'.M) \sim_{(\tau', \tau)\ hfun} \gamma'(M')$.

By restricting the previous lemma, we obtain the correctness of compilation stated as follows.

Theorem 2 (Correctness) *Let* $\emptyset; \emptyset \vdash M : b \rightsquigarrow M'$. $M \downarrow c$ *if and only if* $M' \downarrow c$.

5 Implementation and Measurement

In this section we will discuss some issues on implementation of hole abstractions and then show results of some simple benchmarks. We have implemented some ideas described in this paper in an experimental ML compiler being developed by the author. The compiler was originally developed for the study of a ML compiler based on explicit type parameter passing in [11] and now produces assembly code for the core language of Standard ML extended with structure definitions. All measurements are done on a ALPHA STATION 500/500 with 8 Mbyte cache memory under Digital Unix 4.0A.

So far we have not yet implemented general hole abstractions and the compilation method described in this paper. However, in order to measure how much improvement we can obtain by using hole functions we implemented `happ`, `hcomp`, and several other primitive hole abstractions including `idhfun`, `constail`, and `conshead`. Furthermore, as a phase of the compiler we also implemented the transformation that converts a class of functions producing lists into tail recursive functions as described in Section 2.1.

The elaboration phase of the compiler is modified so that (τ_1, τ_2) *hfun* is treated as a linear type. However, for simplicity we restrict the type system so that polymorphic types are not instantiated by linear types. Although this restriction is not a problem to write examples in this paper, for greater flexibility we should consider a type system which keeps track of use of variables more accurately such as the type system studied by Turner *et al.* [20].

In the intermediate language of the compiler, we represent hole abstractions by a usual record of two pointers to a data structure and a hole. Thus in the intermediate language `('a, 'b) hfun` is represented as follows:

```
type ('a, 'b) hfun = ('a, 'b) prehfun * 'a hole
```

The type `('a, 'b) prehfun` is the type for pointers to values of type `'b` with a hole of type `'a`. The type `'a hole` is the type for holes of type `'a`. In our implementation, `idhfun` is represented by a pair of null pointers which can be distinguished from the pointers to heap memory. By using this representation, the standard optimization of flattening records can optimize manipulations on hole functions. However, by this representation the type system does not prevent using the hole and the data structure part of a hole function independently. Thus it is possible to write a program with side effects on functional data types in the intermediate language.

Our compiler uses tag-free copying garbage collection [19, 11]. There is one problem in implementation of garbage collection in the presence of hole functions: holes point to middle of objects in heap memory. Thus we have to treat holes carefully in implementation of garbage collection. Our strategy is updating the pointers to holes after usual garbage collection is finished. This works because if a hole is live, then there is a live object containing the hole.

We measured the effect of using hole abstractions for several simple programs. The results are shown in Table 1. The columns stand and hfun show execution time for the standard implementation and the implementation using hole abstractions for each program respectively. For `append` and `msort`, we used the automatic transformation converting functions into tail recursive ones described in Section 2.1. In merge sort, the function for merging two lists is converted to a tail recursive function. For flatten we compared the standard implementation using the tail recursive `append` and `hfun_flatten` in Section 2.1. For these functions on lists, the execution time is reduced more than 20%.

To measure effects on programs producing other data types, we used the programs `binsert` and `addone` in Section 2.1. Improvement obtained for `binsert` and `addone` is smaller, but still about 15%. The result for `addone` indicates that optimization by using hole abstractions is useful even if a function cannot be converted to tail recursive one.

6 Related Work

6.1 Linear Types

The calculus we introduced utilizes linearity for efficient implementation of hole application and composition through destructive

	stand	hfun	hfun/stand	Description
`append`	5.46	3.26	0.60	append lists of length 500, 100000 times
`flatten`	2.80	2.22	0.79	flatten a list of length 500 of lists of length 5, 10000 times
`msort`	10.03	7.33	0.73	sort a list of length 500 by merge sort, 10000 times
`binsert`	19.24	15.90	0.83	insert 5000 integers into a binary tree, 1000 times
`addone`	3.85	3.26	0.85	add 1 to each node of a tree of depth 8, 100000 times

Table 1: Execution time (sec)

operations. The linearity we used does not directly correspond to that of linear logic [5], but that for destructive updates used in [21]. However the linearity itself is not sufficient to express data structures with a hole and associated operations. It is because we have to maintain two pointers to a data structure: one is to the top of the data structure and the other is to the hole.

6.2 Tail Recursion Modulo Cons

Several researchers have studied transformation to convert functions which require construction of a data structure using the value of the recursive call into tail recursive functions [9, 24, 3]. This kind of recursion is called tail recursion modulo cons in [24]. In those studies the transformation similar to that in Section 2.1 is formulated as a translation to an imperative language with destructive operations on functional data types. In contrast we can formulate the transformation without introducing explicit destructive operations.

Recently Cheng and Okasaki implemented this optimization for the TIL compiler [3, 18]. They discuss issues related to garbage collection and report similar improvement in their benchmarks.

6.3 Logical Variable

Logical variables are introduced in pure functional programming languages to use programming techniques developed for logic programming languages such as difference-list [10, 15]. Values for logical variables are determined by giving constraints as for variables of logic programming languages. Thus an unconstrained logical variable can be used to represent a hole. Logical variables are usually implemented by using graph reduction. Thus it is not clear that it can be incorporated with call-by-value functional programming languages without significant changes of implementation.

7 Future Work

It is natural to consider data structures with multiple holes: for example, a cons cell where both car and cdr are holes. We think that the extension is possible just by extending hole contexts to the form $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$, though we have not yet checked the details. However, it does not seem easy to develop an efficient method of compilation for the extended calculus.

Another topic for future work is to prove the correctness of compilation based on a low level operational semantics. The semantics we used does not capture sharing of objects and destructive implementation of hole application. We are working on the correctness proof based on a low level semantics similar to the semantics used to prove properties of a language based on linear logic [4].

Acknowledgements

This work is partially supported by Grant-in-Aid for Encouragement of Young Scientists of Japan No. 09780271. We would like to thank Jacques Garrigue, Masahito Hasegawa, Koji Kagawa, Atsushi Ohori, and the anonymous reviewers for their many helpful comments and suggestions.

References

[1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Horth-Holland, 1984.

[2] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24(1):44–67, 1977.

[3] P. Cheng and C. Okasaki. Destination-passing style and generational garbage collection. Unpublished, 1996.

[4] J. Chirimar, C. A. Gunter, and J. G. Riecke. Proving memory management invariants for a language based on linear logic. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 139–147, 1992.

[5] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.

[6] J. C. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *IEEE Symp. on Logic in Computer Science*, 1990.

[7] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11, 1978.

[8] B. Jacobs. Semantics of weakening and contraction. *Annals of Pure and Applied Logic*, 69:73–106, 1994.

[9] J. R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, Computer Science Division, University of California at Berkeley, 1989. USB/CSD 89/502.

[10] G. Lindstrom. Functional programming and the logical variable. In *Proc. ACM Symp. on Principles of Prog. Languages*, pages 266 – 280, 1985.

[11] Y. Minamide. Compilation based on a calculus for explicit type passing. In *Second Fuji International Workshop on Functional and Logic Programming*, pages 301–320. World Scientific, 1996.

[12] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proc. ACM Symp. on Principles of Prog. Languages*, pages 271 – 283, 1996.

[13] A. Ohori. A compilation method for ML-style polymorphic record calculi. In *Proc. ACM Symp. on Principles of Prog. Languages*, 1992.

[14] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transaction on Programming Languages and Systems*, 17(6), 1995.

[15] K. K. Pingali. Lazy evaluation and the logic variable. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 171–198. Addison-Wesley, 1987.

[16] R. Plasmeijer and M. van Eekelen. *Language Report CONCURRENT Clean Version 1.1*. 1996.

[17] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2), December 1975.

[18] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1996.

[19] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 1–11, 1994.

[20] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Functional Programming Languages and Computer Architecture*, 1995.

[21] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.

[22] P. Wadler. Is there a use for linear logic? In *Partial Evaluation and Semantics Based Program Manipulation*, 1991.

[23] P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computing Science*, 1993. LNCS 711.

[24] P. L. Wadler. *Listlessness is Better than Laziness*. PhD thesis, Computer Science Department, Carnegie-Mellon University, 1985. CMU-CS-85-171.

[25] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.