Quasi-Linear Types

Naoki Kobayashi

Department of Information Science, University of Tokyo
email:koba@is.s.u-tokyo.ac.jp

Abstract

Linear types (types of values that can be used just once) have been drawing a great deal of attention because they are useful for memory management, in-place update of data structures, etc.: an obvious advantage is that a value of a linear type can be immediately deallocated after being used. However, the linear types have not been applied so widely in practice, probably because linear values (values of linear types) in the traditional sense do not so often appear in actual programs. In order to increase the applicability of linear types, we relax the condition of linearity by extending the types with information on an evaluation order and simple dataflow information. The extended type system, called a quasi-linear type system, is formalized and its correctness is proved. We have implemented a prototype type inference system for the core-ML that can automatically find out which value is linear in the relaxed sense. Promising results were obtained from preliminary experiments with the prototype system.

1 Introduction

1.1 Linear types

A number of type systems based on Girard's linear logic [6] have been proposed [1, 3, 9, 13, 14, 22]. They guarantee that certain data structures (called *linear* values) are accessed just once (or at most once). The distinction between linear values and other values brings us several benefits: improvement of memory management [1], safe inlining [22], etc. Among them, we are here interested in the improvement of memory management. If some data structure is statically known to be linear, it can be deallocated immediately after it is accessed. Moreover, if another heap value needs to be created after a linear value is accessed, we can just replace the linear value with the new value, instead of deallocating the linear value and allocating space for the new data (provided that the physical size of the new value is not greater than that of the linear value). For instance (throughout this paper, we assume a call-by-value language), when the function $\lambda\langle x, y\rangle.\langle x, y+1\rangle$ is called, if the argument is a linear pair, the second element y of the pair can be destructively updated to $y+1$. Similarly, the "append" of linear lists can be performed destructively, without copying cons cells. This kind of improvement will be especially effective for functional programs, because many intermediate data structures like cons cells and closures are created in naive functional programs.

1.2 Limitation of linear types

In spite of the above-mentioned benefit from linear type systems, they do not seem to have been used so widely, even after type inference algorithms [8, 9, 22] were proposed that can automatically find which values are linear.

We think one of the major reasons for this is that linear type systems are too naive for the above application: what is actually important for automatic deallocation and in-place update is to identify which is the last access to a heap value; the condition of linearity is too strong for this purpose. For example, consider an expression $M \equiv \textbf{let } x = 2.0 \textbf{ in let } y = x + 1.0 \textbf{ in } y + x$: it is easy to see that $y + x$ is the last access to x, but linear type systems cannot insert a code to deallocate x — just because x is used twice! This limitation also forces a programmer to follow a particular programming style. For instance, in $\lambda z.\langle \textbf{fst}(z), \textbf{snd}(z) + 1\rangle$, the pair z is judged to be accessed twice; so, if one wants the pair to be deallocated, one must write $\lambda\langle x, y\rangle.\langle x, y+1\rangle$ instead.

1.3 Our proposal — almost linear types (quasi-linear types)

In order to remove the above-mentioned limitation of linear types, we relax the condition of linearity.

A key idea is to express information on the evaluation order and dataflow by using types. Recall the above expression M: if a type system can take it into account that $x + 1.0$ is evaluated before $y + x$, it can ignore the use of x in $x + 1.0$ and treat the above expression in the same way as $\textbf{let } x = 2.0 \textbf{ in let } y = 1.0 \textbf{ in } y + x$. In order to obtain such information on the evaluation order, the type system should also be able to deal with some dataflow information. Consider, for example, an expression $\textbf{let } y = f(x) \textbf{ in } g(x) + h(y)$. $f(x)$ is evaluated before $g(x)$ but it does not imply that the access to x in $g(x)$ is the last one: x may be returned by $f(x)$ and accessed again in $h(y)$. In order to deal with this, we distinguish between the type of a value that must be used up in an expression and

POPL 99 San Antonio Texas USA

the type of a value that may be returned as a part of the evaluation result of an expression.

It would be possible to obtain the above kinds of information by using more sophisticated analyses, but we instead obtain it by a very simple method — by introducing a new *use* [9, 22]. Advantages of that approach include: (1) the whole analysis can be formalized uniformly in terms of a type system (this is especially advantageous when we introduce polymorphism) and (2) a program containing free variables can be analyzed as long as their type information is provided, which is useful for modular (or incremental) analysis and separate compilation of programs. We first review the ideas of previous linear type systems [9, 22] and then overview the ideas of our system below.

1.3.1 Review of linear type systems

A *use* was introduced to control how often values can be accessed [9, 22]: it was either 0 (not used at all), 1 (accessed at most once), or ω (accessed an arbitrary number of times) in [9] and was either 1 or ω in [22]. Types are annotated with such uses. For example, $real^1$ represents the type of real numbers that can be accessed at most once, and $int \to^{\omega} int$ is the type of functions on integers that can be called an arbitrary number of times. A type judgment can be accordingly refined so that it gives more useful information than the ordinary one: for example, $x : real^1 \vdash N : real^{\omega}$ means not only that x is used as a real number in N but also that it is used at most once. So, it is invalid if N is **let** $y = x + 1.0$ **in** $x + y$.

The following type derivation highlights a key point of the linear type system:

$$\frac{\begin{array}{c} x : real^1 \vdash x + 1.0 : real^1 \\ x : real^1, y : real^1 \vdash x + y : real^1 \end{array}}{\begin{array}{l} x : real^1 + x : real^1 (= x : real^{\omega}) \\ \qquad \vdash \textbf{let } y = x + 1.0 \textbf{ in } x + y : real^1 \end{array}}$$

The premises of the derivation imply that x is used once each in $x + 1.0$ and $x + y$. In order to know how x is totally used in the expression, we combine the type environments of subexpressions by *adding* the corresponding uses as above. This is in contrast to the following rule of ordinary type systems, where type environments are *shared* between subexpressions:

$$\frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, y : \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \textbf{let } y = M_1 \textbf{ in } M_2 : \tau_2}$$

By using the linear type system, we can infer how often each heap value is accessed, and annotate each allocation with a use. For instance, we can annotate the expression M as **let** $x^{\omega} = 2.0$ **in let** $y^1 = x + 1.0$ **in** $x + y$, so that y is deallocated after it is accessed in $x + y$ while x is not deallocated.

1.3.2 Overview of our type system

In order to express simple dataflow information, we introduce another use δ. Intuitively, a value with use δ (which we call a *δ-value* below) may be accessed many times but only *locally* — it cannot be returned as (a part of) the computed result. So, x can have type $real^{\delta}$ in $x + 1.0$ and $(\lambda y.(y + 1.0))x$, but not in $\langle x, 1.0 \rangle$ or $\lambda y.x + y$. A value with use 1 (hereafter, we call it a *quasi-linear* value or simply a *linear* value, and call a use-once value in the traditional sense a *strictly linear* value) can now be accessed in a more relaxed manner: It can be accessed many times as a δ-value, and then it can be accessed as a strictly linear value (i.e., as a value that can be accessed once and deallocated).

In order to guarantee such usage of a value, it is crucial that the type system can take the evaluation order into account. The type derivation shown above is now changed as follows:

$$\frac{\begin{array}{c} x : real^{\delta} \vdash x + 1.0 : real^1 \\ x : real^1, y : real^1 \vdash x + y : real^1 \end{array}}{\begin{array}{l} x : real^{\delta}; x : real^1 (= x : real^1) \\ \qquad \vdash \textbf{let } y = x + 1.0 \textbf{ in } x + y : real^1 \end{array}}$$

Here, the new combination $(x : real^{\delta}); (x : real^1)$ of type environments captures the fact that x is first accessed as a δ-value in $x + 1.0$, and then it is accessed as a linear value in $x + y$. Thus, x can be deallocated after $x + y$ is evaluated.

Because a quasi-linear value may be used more than once, in addition to allocation of a heap value, each access to a heap value is also annotated with a use: it is for distinguishing between the last access of the value and the other accesses. For example, the expression M is annotated as **let** $x^1 = 2.0$ **in let** $y^1 = x^{\delta} + 1.0$ **in** $y^1 + x^1$ (actually, we will annotate constructors and destructors of heap values instead of variables). Here, we annotate x in $y + x$ with 1 in order to indicate that it is the last access, while x in $x + 1.0$ is annotated with δ in order to indicate that it is not the last one.

1.4 Main results

Main contributions of the present work are formalization of the new type system sketched above and a proof of its correctness. Since the evaluation order must be taken into account, the proof is non-trivial and more involved than that of previous type systems.

Another contribution is implementation of a type inference system for the core-ML (i.e., Standard ML without modules [15]) based on the new type system, which inputs an unannotated core-ML expression and outputs a use-annotated expression. We have so far tested several fairly small programs and obtained promising results: for programs that use lists frequently (such as sorting programs, the sieve of Eratosthenes and Conway's game of life), the type inference system could judge that most heap values (in the case of the sorting programs and the sieve of Eratosthenes, all values except for top-level functions) are linear, which indicates that they can be executed almost without garbage collection. Moreover, the annotated programs output by the system indicate that most linear cons cells in those programs can be updated in-place (the application of linear types to the in-place quick sort has been suggested by Baker [2] but a remarkable point here is that it can be performed for naive programs with no programmer's annotation).

1.5 Structure of the paper

The rest of this paper is structured as follows. Section 2 introduces the syntax of the target language. Section 3 gives our type system for judging whether an expression is correctly annotated with uses. Section 4 sketches a proof of

the correctness of the type system and Section 5 briefly discusses type inference issues. Section 6 discusses several extensions of the target language. Section 7 explains our prototype type inference system and the results of preliminary experiments. Section 8 discusses related work and Section 9 concludes this paper. For space restriction, we omit some formal definitions, proofs, etc.: they are found in the full version of this paper [12].

2 Target Language

2.1 Uses

As explained in Section 1, a *use* is introduced to control how often and in which way each heap value can be accessed.

Definition 2.1 [uses]: A use is 0, δ, 1, or ω.

We use metavariables $\kappa_1, \kappa_2, \ldots$ for uses.

A new point here is the use δ: basically, we say that a heap value is accessed in an expression as a value of use δ if it is accessed *locally* inside the expression and cannot be returned as a part of the evaluation result. 0 means that the data cannot be accessed at all, 1 means that after the data is accessed in a restricted manner as a value of use δ, it can be either (1) accessed at most once and deallocated inside the current expression being evaluated or (2) put into the evaluation result as a value of use 1. ω means that the data can be accessed many times in any way.

A value of use ω can be regarded as a value of any other use, and a value of 1 can be regarded as a value of use δ or 0. In order to express this kind of relationship between uses, we define a total order $\leq$ on uses by $0 \leq \delta \leq 1 \leq \omega$. We write $\kappa_1 < \kappa_2$ if $\kappa_1 \neq \kappa_2$ and $\kappa_1 \leq \kappa_2$.

Remark 2.2: By a heap value being *accessed* (or *used*), we mean the contents of the heap value are indeed read; so, just passing the reference to the heap value does not count as access. For example, we do not say that y is accessed in $(\lambda x.x)y$, since the expression can be reduced to y without reading the actual contents of y. A pair is considered to be accessed only when its first or second element is extracted, a closure is accessed only when it is invoked, and a real number is accessed only when it is passed as an argument to a primitive function on real numbers.

2.2 Expressions

We consider a simply-typed λ-calculus with recursion and pairs as a target language. Its extension with polymorphism and other data structures will be briefly discussed in Section 6. Because our type system is sensitive to an evaluation order, we use the K-normal form [5] in order to make the evaluation order explicit and name each intermediate value. (For readability, we sometimes use expressions that are not in K-normal form when giving examples.)

The syntax of expressions is given in Figure 1. Each allocation or read operation of a heap value is annotated with a use. The annotation can be automatically inferred by a type inference algorithm explained in Section 5. We write $\mathcal{E}$ for the set of expressions.

The metavariable V represents a value that can be represented in one word and need not be allocated in the heap. Here, it is either an integer constant (denoted by n) or a variable (denoted by x). A variable can be considered a heap address.

$\langle V_1, V_2 \rangle^\kappa$ and $\lambda^\kappa y.M_1$ allocate a pair and a closure respectively in the heap, and records that their use is κ (which shall be restricted to $0, 1$, or ω by the type system given later). The recorded use expresses how the value can be accessed in the rest of the computation: 0 means that the value cannot be accessed at all (so, $\langle V_1, V_2 \rangle^0$ need not actually allocate the pair in the heap), 1 means that the value can be accessed as a linear value and may be deallocated later, and ω means that the value can be accessed in an arbitrary manner and can be deallocated only by other mechanisms such as garbage collection. $\mathbf{rec}^\kappa(f, y, M_1)$ creates a recursive function f such that $f(y) = M_1$.

There are three operations to access the heap: $\mathbf{fst}^\kappa(x)$ ($\mathbf{snd}^\kappa(x)$, resp.) extracts the first (second, resp.) element of the pair stored at x, and $apply^\kappa(y, V)$ reads the closure stored at y and applies it to V. κ attached to an access operation represents as a value of what use the heap value is accessed by this operation. If κ is δ, the heap is just read and unchanged; if it is 1, the heap value is deallocated after being read (if the use of the heap value is also 1[1]). If κ is ω, then the operation assumes that the accessed heap value is an ω-value and does not deallocate the value. (so, ω is the same as δ in effect and is unnecessary; it is included just for technical convenience). If κ is greater than the actual use recorded at the accessed address, the access is invalid and causes an error.

A conditional expression **if0** V **then** M_1 **else** M_2 is reduced to M_1 if $V = 0$ and otherwise reduced to M_2.

Example 2.3: **let** $x = \langle 1, 2 \rangle^1$ **in let** $y = \mathbf{fst}^\delta(x)$ **in let** $z = \mathbf{snd}^1(x)$ **in** z allocates a linear pair $\langle 1, 2 \rangle$ in the heap, and then reads its first element; at this moment, x is not deallocated since **fst** is annotated with δ. After that, the second element is read and the pair is deallocated.

2.3 Operational semantics

In order to clarify how use annotation is used for allocating/deallocating heap values at run-time, we define an operational semantics as a rewriting relation on pairs of a heap and an expression [17, 22].

Definition 2.4 [evaluation contexts]: The set of *evaluation contexts* is given by the following syntax:

$$C[\,] ::= [\,] \mid \mathbf{let}\ x = C[\,]\ \mathbf{in}\ M.$$

We write $C[M]$ for the expression obtained from $C[\,]$ by replacing the hole $[\,]$ with M.

Definition 2.5 [heap]: A *heap value*, denoted by h, is a term of the form $\langle V_1, V_2 \rangle$ or $\lambda x.M$. A *heap* is a mapping from a finite set of variables to pairs consisting of a use and a heap value. We write $\{x_1 \mapsto^{\kappa_1} h_1, \ldots, x_n \mapsto^{\kappa_n} h_n\}$ for a heap H such that $H(x_i) = (\kappa_i, h_i)$.

The use associated with a heap value is used for judging whether the value can be deallocated after it is accessed as a linear value: the value can be deallocated only if the use is 1. The use 0 means that the value has been already

[1] The check of the use of the heap value is necessary, because we allow an ω-value to be accessed as a linear value. If we forbid such coercion of an ω-value into a linear value as in [22], then the check is unnecessary. However, we prefer not to make such restriction because the check can be implemented efficiently(see Remark 2.6).

$$
\begin{array}{rcl}
M \text{ (expressions)} & ::= & V \mid \mathbf{let}\ x = \langle V_1, V_2\rangle^{\kappa}\ \mathbf{in}\ M \mid \mathbf{let}\ x = \mathbf{fst}^{\kappa}(y)\ \mathbf{in}\ M \mid \mathbf{let}\ x = \mathbf{snd}^{\kappa}(y)\ \mathbf{in}\ M \\
 & & \mid \mathbf{let}\ x = \lambda^{\kappa} y.M_1\ \mathbf{in}\ M_2 \mid \mathbf{let}\ x = \mathbf{rec}^{\kappa}(f, y, M_1)\ \mathbf{in}\ M_2 \mid \mathbf{let}\ x = \mathit{apply}^{\kappa}(y, V)\ \mathbf{in}\ M \\
 & & \mid \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 \mid \mathbf{if0}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 \\
V \text{ (non-heap values)} & ::= & x \mid n
\end{array}
$$

Figure 1: The Syntax of Expressions

deallocated. We often write h^{κ} for $\langle V_1, V_2\rangle^{\kappa}$ or $\lambda^{\kappa} x.M$. We write $\mathcal{H}$ for the set of heaps.

The reduction relation $\rightsquigarrow \in (\mathcal{H}\times\mathcal{E}) \times ((\mathcal{H}\times\mathcal{E}) \cup \{\mathbf{Error}\})$ is defined by the rules given in Figure 2. **Error** indicates that invalid access to the heap has occurred (there are other kinds of errors such as application of a non-function to a value, etc.; however, because the lack of those errors can be guaranteed by a type system in the usual way, this paper focuses on errors caused by invalid heap access).

The first two rules (R-HEAP) and (R-REC) are for allocating heap values. The use κ is recorded at a new address together with the allocated value. The heap is not changed in (R-SVAL) since V is a one-word value.

The rules (R-APP) and (R-APP-ERROR) for function application deserve attention. Since $\mathit{apply}^{\kappa'}(x, V)$ reads a closure stored at x as a value with use κ', the currently available use κ of x should be greater than or equal to κ': otherwise, the application causes an error. If $\kappa = \kappa' = 1$, the closure is deallocated after the application and it can no longer be called; so, the use of x should be changed to 0. It is expressed by the subtraction $\kappa - \kappa'$, whose definition is given in Figure 3. Similarly, access to a pair $\mathbf{fst}^{\kappa'}(x)$ or $\mathbf{snd}^{\kappa'}(x)$ succeeds only if $\kappa \geq \kappa'$, and the use is decreased by κ' after the access ((T-FST) — (T-SND-ERROR)).

Remark 2.6: One may think that the above operation on uses incurs a heavy overhead at run-time. However, because the type system guarantees that no invalid access occurs, the check of $\kappa \geq \kappa'$ is always guaranteed to succeed and can be eliminated. Moreover, the use associated with a heap value can change only from 1 to 0; since this change means that the heap value is deallocated in the actual implementation, the use need not be updated actually. This further implies that the use of a heap value can be recorded not in the heap space itself but in the pointer to it by using a one-bit tag, and that the extra run-time cost is only to check this tag (to know whether the use of the accessed heap value is 1 or ω) when the access operation is annotated with 1. So, we think the actual run-time overhead is quite small.

Example 2.7: The expression in Example 2.3 is reduced as follows:

$$
\begin{array}{ll}
 & (\{\}, \mathbf{let}\ x = \langle 1,2\rangle^{1}\ \mathbf{in} \\
 & \quad \mathbf{let}\ y = \mathbf{fst}^{\delta}(x)\ \mathbf{in}\ \mathbf{let}\ z = \mathbf{snd}^{1}(x)\ \mathbf{in}\ z) \\
\rightsquigarrow & (\{u \mapsto^{1} \langle 1,2\rangle\}, \\
 & \quad \mathbf{let}\ y = \mathbf{fst}^{\delta}(u)\ \mathbf{in}\ \mathbf{let}\ z = \mathbf{snd}^{1}(u)\ \mathbf{in}\ z) \\
\rightsquigarrow & (\{u \mapsto^{1} \langle 1,2\rangle\}, \mathbf{let}\ y = 1\ \mathbf{in}\ \mathbf{let}\ z = \mathbf{snd}^{1}(u)\ \mathbf{in}\ z) \\
\rightsquigarrow & (\{u \mapsto^{1} \langle 1,2\rangle\}, \mathbf{let}\ z = \mathbf{snd}^{1}(u)\ \mathbf{in}\ z) \\
\rightsquigarrow & (\{u \mapsto^{0} \langle 1,2\rangle\}, \mathbf{let}\ z = 2\ \mathbf{in}\ z) \\
\rightsquigarrow & (\{u \mapsto^{0} \langle 1,2\rangle\}, 2)
\end{array}
$$

The heap value u is considered to be deallocated when the use changes from 1 to 0.

Example 2.8: If the expression in Example 2.3 is wrongly annotated as:

$$\mathbf{let}\ x = \langle 1,2\rangle^{1}\ \mathbf{in}\ \mathbf{let}\ y = \mathbf{fst}^{1}(x)\ \mathbf{in}\ \mathbf{let}\ z = \mathbf{snd}^{\delta}(x)\ \mathbf{in}\ z,$$

then it is reduced to **Error** as follows:

$$
\begin{array}{ll}
 & (\{\}, \mathbf{let}\ x = \langle 1,2\rangle^{1}\ \mathbf{in} \\
 & \quad \mathbf{let}\ y = \mathbf{fst}^{1}(x)\ \mathbf{in}\ \mathbf{let}\ z = \mathbf{snd}^{\delta}(x)\ \mathbf{in}\ z) \\
\rightsquigarrow & (\{u \mapsto^{1} \langle 1,2\rangle\}, \\
 & \quad \mathbf{let}\ y = \mathbf{fst}^{1}(u)\ \mathbf{in}\ \mathbf{let}\ z = \mathbf{snd}^{\delta}(u)\ \mathbf{in}\ z) \\
\rightsquigarrow & (\{u \mapsto^{0} \langle 1,2\rangle\}, \mathbf{let}\ y = 1\ \mathbf{in}\ \mathbf{let}\ z = \mathbf{snd}^{\delta}(u)\ \mathbf{in}\ z) \\
\rightsquigarrow & (\{u \mapsto^{0} \langle 1,2\rangle\}, \mathbf{let}\ z = \mathbf{snd}^{\delta}(u)\ \mathbf{in}\ z) \\
\rightsquigarrow & \mathbf{Error}
\end{array}
$$

3 Type System

As was shown in Example 2.8, if an expression is wrongly annotated with uses, a run-time error may occur. In order to reject such expressions, we introduce a type system. Our type system is a conservative extension of the usual type system for the simply typed λ-calculus in the sense that any expressions well-typed in the usual type system are well typed also in our type system.

There are two major differences between our quasi-linear type system and the usual linear type systems. As explained in Section 1, a linear value can be accessed many times as long as the last access is identified by the quasi-linear type system. The other difference is the interpretation of a pair type (see below).

3.1 Types

Definition 3.1 [types]: The set of types, ranged over by τ, is given by the following syntax.

$$\tau \ ::= \ \mathit{int} \mid \tau_1 \times^{\kappa} \tau_2 \mid \tau_1 \rightarrow^{\kappa_1,\kappa_2} \tau_2$$

κ of $\tau_1 \times^{\kappa} \tau_2$ and κ_1 of $\tau_1 \rightarrow^{\kappa_1,\kappa_2} \tau_2$ expresses how a pair and a closure can be accessed. We require κ_2 of $\tau_1 \rightarrow^{\kappa_1,\kappa_2} \tau_2$ not to be δ. It represents how often (0, 1, or ω) the closure can be invoked in the sense of previous linear type systems [9, 22]. The closure itself is allocated and deallocated based on κ_1. The reason why we keep κ_2 will become clear when we present a typing rule for closure creations.

Note that the meaning of a pair type is different from that in previous linear type systems [8, 22]. In the previous linear type systems, $\mathit{int} \times^{\omega} (\mathit{int} \times^{1} \mathit{int})$ corresponds to the linear logic formula $!(\mathit{int} \otimes (\mathit{int} \otimes \mathit{int}))$; so, if a pair $\langle 1, \langle 2,3\rangle\rangle$ has this type, it can be read an arbitrary number of times, and *each time* the second element $\langle 2,3\rangle$ is extracted, it can be read at most once. On the other hand, in our quasi-linear type system, the second element $\langle 2,3\rangle$ can be accessed only linearly *throughout all the access* to the pair $\langle 1, \langle 2,3\rangle\rangle$ (not each time $\langle 2,3\rangle$ is extracted).

$$(H, C[\textbf{let } x = h^{\kappa} \textbf{ in } M]) \rightsquigarrow (H\{y \mapsto^{\kappa} h\}, C[[y/x]M]) \ (y \text{ fresh}) \quad \text{(R-HEAP)}$$

$$(H, C[\textbf{let } x = \textbf{rec}^{\kappa}(f, y, M_1) \textbf{ in } M]) \rightsquigarrow (H\{z \mapsto^{\kappa} \lambda y.[z/f]M_1\}, C[[z/x]M]) \ (z \text{ fresh}) \quad \text{(R-REC)}$$

$$(H, C[\textbf{let } x = V \textbf{ in } M]) \rightsquigarrow (H, C[[V/x]M]) \quad \text{(R-SVAL)}$$

$$\frac{\kappa \geq \kappa' > 0}{(H\{x \mapsto^{\kappa} \lambda y.M\}, C[apply^{\kappa'}(x, V)]) \rightsquigarrow (H\{x \mapsto^{\kappa-\kappa'} \lambda y.M\}, C[[V/y]M])} \quad \text{(R-APP)}$$

$$\frac{(\kappa < \kappa') \vee (\kappa' = 0)}{(H\{x \mapsto^{\kappa} \lambda y.M\}, C[apply^{\kappa'}(x, V)]) \rightsquigarrow \textbf{Error}} \quad \text{(R-APP-ERROR)}$$

$$\frac{\kappa \geq \kappa' > 0}{(H\{x \mapsto^{\kappa} \langle V_1, V_2\rangle\}, C[\textbf{fst}^{\kappa'}(x)]) \rightsquigarrow (H\{x \mapsto^{\kappa-\kappa'} \langle V_1, V_2\rangle\}, C[V_1])} \quad \text{(R-FST)}$$

$$\frac{(\kappa < \kappa') \vee (\kappa' = 0)}{(H\{x \mapsto^{\kappa} \langle V_1, V_2\rangle\}, C[\textbf{fst}^{\kappa'}(x)]) \rightsquigarrow \textbf{Error}} \quad \text{(R-FST-ERROR)}$$

$$\frac{\kappa \geq \kappa' > 0}{(H\{x \mapsto^{\kappa} \langle V_1, V_2\rangle\}, C[\textbf{snd}^{\kappa'}(x)]) \rightsquigarrow (H\{x \mapsto^{\kappa-\kappa'} \langle V_1, V_2\rangle\}, C[V_2])} \quad \text{(R-SND)}$$

$$\frac{(\kappa < \kappa') \vee (\kappa' = 0)}{(H\{x \mapsto^{\kappa} \langle V_1, V_2\rangle\}, C[\textbf{snd}^{\kappa'}(x)]) \rightsquigarrow \textbf{Error}} \quad \text{(R-SND-ERROR)}$$

$$(H, C[\textbf{if0 } 0 \textbf{ then } M_1 \textbf{ else } M_2]) \rightsquigarrow (H, C[M_1]) \quad \text{(R-IFT)}$$

$$\frac{n \neq 0}{(H, C[\textbf{if0 } n \textbf{ then } M_1 \textbf{ else } M_2]) \rightsquigarrow (H, C[M_2])} \quad \text{(R-IFF)}$$

Figure 2: Operational Semantics

Example 3.2: $(int \times^{\delta} int) \rightarrow^{1,\omega} int$ is the type of a linear closure that takes a pair of integers as an argument, uses it locally, and returns an integer. The function may be invoked many times (thus its use is ω in the strict sense), but it can be invoked only linearly in the relaxed sense (i.e., invoked as a closure of use δ except for the last invocation). For example, $\lambda^1 x.(\textbf{fst}^{\delta}(x) + \textbf{snd}^{\delta}(x))$ can have this type.

Example 3.3: Consider an expression
let $f = \lambda^1 z.(\textbf{fst}^{\delta}(x) + z)$ **in let** $y = \textbf{fst}^1(\textbf{snd}^{\delta}(x))$ **in** M
and suppose x does not appear in M. Because the second element of the pair x is accessed just once in $\textbf{fst}^1(\textbf{snd}^{\delta}(x))$, it can be assigned the type $int \times^{\omega} (int \times^1 int)$; so, the pair of integers can be deallocated at $\textbf{fst}^1$. Consider another expression **let** $f = \lambda^1 z.(\textbf{fst}^1(\textbf{snd}^1(z)) + 1)$ **in** $apply^{\delta}(f, x) + apply^1(f, x)$. The second element of x is accessed once (by $\textbf{fst}^1$) each time it is extracted by $\textbf{snd}^1(z)$. Since it is totally accessed more than once, the type $int \times^{\omega} (int \times^{\omega} int)$ is assigned to x in our quasi-linear type system.

3.2 Type judgment

A type judgment is of the form $\Gamma \vdash M : \tau$, where Γ (called a type environment) is a mapping from a finite set of variables to types. It expresses how each heap value is accessed during evaluation of M. For example, $x : int \times^{\delta} int \vdash M : int \times^1 int$ implies (1)M may access the heap address x, (2)the result is a linear pair of integers, and (3)x is used up in M and does not escape through the result.

Notation 3.4: We write $V_1 : \tau_1, \ldots, V_n : \tau_n$ for the type environment Γ such that $dom(\Gamma) = \{V_1, \ldots, V_n\} \cap \mathcal{V}$ and $\Gamma(V_i) = \tau_i$ for each $V_i \in dom(\Gamma)$, where $\mathcal{V}$ denotes the set of variables. For example, $(x : int, 2 : int)$ denotes the type environment that maps x to int. If $V \notin dom(\Gamma)$, we write $\Gamma, V : \tau$ for the type environment Γ' such that (1) $\Gamma' = dom(\Gamma) \cup (\{V\} \cap \mathcal{V})$ and (2) $\Gamma'(x) = \tau$ if $x \in (\{V\} \cap \mathcal{V})$ and $\Gamma'(x) = \Gamma(x)$ if $x \in dom(\Gamma)$.

3.3 Operations on uses, types, and type environments

In presenting typing rules, we use several operations on uses, types, and type environments given in Figure 3 and 4. As explained in Section 1, the operation $\kappa_1 + \kappa_2$ is used for computing the total use of a value when it is accessed as κ_1 in one place and as κ_2 in another place. When the order of the uses is statically known, $\kappa_1; \kappa_2$ is used instead. $\lceil \kappa \rceil$ is the least non-δ use that is greater than or equal to κ, and $\lfloor \kappa \rfloor$ is the greatest non-δ use that is less than or equal to κ. $\kappa_1 \cdot \kappa_2$ is used for computing the total use of a value when the value is used as a κ_2-value κ_1 times. Motivations for these operations will become clearer when typing rules are given below.

These operations are extended to operations on types and type environments. For example, if a heap value is accessed as a value of type $int \times^{\delta} int$ in one place and as a value of type $int \times^1 int$ in another place, it is totally accessed as a value of type $(int \times^{\delta} int) + (int \times^1 int) = int \times^{\omega} int$, which means that the value may be accessed more than once.

The definitions of operations on pair types and function types deserve special attention; notice that the operations act on sub-components of pair type, but not on the arguments and return types of functions. In order to understand the reason for this, suppose x is accessed in two places, each as a value of type $(int \times^1 int) \times^1 int$ (i.e., the inner pair of integers and the outer pair are both accessed linearly). Then, both the inner pair of integers and the outer pair are totally accessed more than once; it is expressed by the type $(int \times^{1+1} int) \times^{1+1} int$ obtained by adding each use attached to the pair types, not by the type $(int \times^1 int) \times^{1+1} int$ (which means that the inner pair is totally accessed only linearly). On the other hand, suppose x is accessed in two places as a function of type $(int \times^1 int) \rightarrow^{1,1} (int \times^1 int)$. Then, the function is invoked twice, and each time it is invoked, it uses an argument pair linearly and returns a linear pair. So, the total access to the function is expressed by $(int \times^1 int) \rightarrow^{1+1,1+1} (int \times^1 int)$,

not by $(int \times^{1+1} int) \to^{1+1,1+1} (int \times^{1+1} int)$. Similarly, the operation $\lceil \cdot \rceil$ also acts on sub-components of pair type but not on the arguments and return types of function types. This is because $\lceil \tau \rceil$ is intended to express the type of an expression whose evaluation result does not contain δ-values. Although the type $(int \times^{\delta} int) \to^{1,1} (int \times^{1} int)$ contains δ, it just means that a function of that type uses an argument pair locally, not that the function must be used locally. So, $\lceil (int \times^{\delta} int) \to^{1,1} (int \times^{1} int) \rceil = (int \times^{\delta} int) \to^{1,1} (int \times^{1} int)$.

We often omit $\cdot$ and write $\kappa_1\kappa_2$ for $\kappa_1 \cdot \kappa_2$ and $\kappa\Gamma$ for $\kappa \cdot \Gamma$. We give higher precedence to $\cdot$, $+$, and ; in this order.

Example 3.5: Let τ_1 be $int \times^{\delta} (int \times^{\delta} int)$ and τ_2 be $int \times^{1} (int \times^{0} int)$. Then,

$$\begin{array}{lcl} \tau_1 + \tau_2 & = & int \times^{\omega} (int \times^{\delta} int) \\ \tau_1; \tau_2 & = & int \times^{1} (int \times^{\delta} int) \\ \lceil \tau_1 \rceil & = & int \times^{1} (int \times^{1} int) \\ \omega \cdot \tau_1 & = & int \times^{\omega} (int \times^{\omega} int). \end{array}$$

$$\begin{array}{l} (x : \tau_1, y : \tau_1) + (x : \tau_2) \\ \quad = x : (\tau_1 + \tau_2), y : \tau_1 \\ \quad = x : int \times^{\omega} (int \times^{\delta} int), y : int \times^{\delta} (int \times^{\delta} int). \end{array}$$

3.4 Typing rules

3.4.1 Variables, constants, and weakening

The rule for variables and constants is:

$$V : \tau \vdash V : \tau \qquad \text{(T-VAL)}$$

If an expression is well typed under some type environment, then it should also be well typed under a type environment that represents more access capabilities. For example, if $x : int \times^{1} int \vdash M : int$, then $x : int \times^{\omega} int \vdash M : int$, since the former judgment requires x to be used as a linear pair, while the latter does not impose such a requirement. The following rule allows such weakening of an assumption:

$$\frac{\Gamma' \vdash M : \tau \qquad \Gamma' \leq \Gamma}{\Gamma \vdash M : \tau} \qquad \text{(T-WEAK)}$$

The relation $\Gamma' \leq \Gamma$, which means "Γ represents more access capabilities than Γ' (in other words, Γ allows more liberal access to the heap)," is defined in Figure 4.

3.4.2 Allocation of heap values

Let us consider an expression $\mathbf{let}\ x = \langle y, z \rangle^{\kappa}\ \mathbf{in}\ M$. In M, y and z can be accessed in two ways: either directly by using addresses y and z, or through the pair x (by extracting y and z). If $\Gamma, x : \tau_1 \times^{\kappa} \tau_2 \vdash M : \tau$, then the access in the former way is represented by the types $\Gamma(y)$ and $\Gamma(z)$, while the access in the latter way is represented by the types τ_1 and τ_2. Therefore, information on the total access is obtained by adding those types. For example, if $\Gamma(y) = int \times^{1} int$ and $\tau_1 = int \times^{1} int$, then the total access to y can be represented by $\Gamma(y) + \tau_1 = int \times^{\omega} int$. Thus, the rule for the allocation of a pair is:

$$\frac{\Gamma, x : \tau_1 \times^{\kappa} \tau_2 \vdash M : \tau_3 \qquad \kappa \neq \delta}{\Gamma + V_1 : \tau_1 + V_2 : \tau_2 \vdash \mathbf{let}\ x = \langle V_1, V_2 \rangle^{\kappa}\ \mathbf{in}\ M : \tau_3} \qquad \text{(T-PAIR)}$$

We require $\kappa \neq \delta$ because there is no use creating a δ-value: since it can never be deallocated, creating a δ-value is the same as creating an ω-value.

Remark 3.6: Actually, annotating allocation of a pair with δ is meaningful and may be rather useful if the type τ_3 of M does not contain δ: we can modify the operational semantics so that $\mathbf{let}\ x = \langle V_1, V_2 \rangle^{\delta}\ \mathbf{in}\ M$ allocates the pair $\langle V_1, V_2 \rangle$ in the stack and deallocates it after M has been evaluated (recall that a value with use δ cannot be a part of the evaluation result).

The rule for the allocation of a closure is:

$$\frac{\Gamma_1, x : \tau_1 \vdash M_1 : \lceil \tau_2 \rceil \qquad \Gamma_2, f : \tau_1 \to^{\kappa_1, \kappa_2} \lceil \tau_2 \rceil \vdash M_2 : \tau_3 \qquad \kappa_1 \neq \delta}{\kappa_2 \lceil \Gamma_1 \rceil + \Gamma_2 \vdash \mathbf{let}\ f = \lambda^{\kappa_1} x.M_1\ \mathbf{in}\ M_2 : \tau_3} \qquad \text{(T-ABS)}$$

$\lceil \tau_2 \rceil$ in the premise $\Gamma_1, x : \tau_1 \vdash M_1 : \lceil \tau_2 \rceil$ guarantees that when the closure $\lambda x.M$ is called, it does not return a δ-value as a result. $\Gamma_1, x : \tau_1 \vdash M_1 : \lceil \tau_2 \rceil$ also means that during each call of the closure, heap values are accessed as described by Γ_1. Because the other premise means that the closure will be called κ_2 times, the total access to heap values by M_1 can be represented by κ_2 copies of Γ_1, which is written as $\kappa_2\Gamma_1$. Since the access happens only later (not when $\lambda x.M_1$ is evaluated and the closure is created), $\lceil \_ \rceil$ is applied to it, so that it does not contain δ-values.

The following rule for recursive functions is similar to (T-ABS) except for the estimation of the use of the created function:

$$\frac{\Gamma_1, x : \tau_1, f : \tau_1 \to^{\kappa_1, \kappa_2} \lceil \tau_2 \rceil \vdash M_1 : \lceil \tau_2 \rceil \qquad \Gamma_2, f : \tau_1 \to^{\kappa_3, \kappa_4} \lceil \tau_2 \rceil \vdash M_2 : \tau_3}{\kappa_4(1 + \kappa_2)\lceil \Gamma_1 \rceil + \Gamma_2 \vdash \mathbf{let}\ f = \mathbf{rec}^{\lceil \kappa_3(1+\kappa_1) \rceil}(f, x, M_1)\ \mathbf{in}\ M_2 : \tau_3} \qquad \text{(T-REC)}$$

The total number of calls of the function f is calculated by using the number κ_2 of calls in M_1 and the number κ_4 of calls in M_2. Since each invocation of f may result in other κ_2 invocations of f, the total calls are counted as $\kappa_4(1 + \kappa_2 + \kappa_2^2 + \cdots) = \kappa_4(1 + \kappa_2)$. The use of function f itself (in the relaxed sense) is similarly calculated by using its uses κ_1 and κ_3 in M_1 and M_2. The ceiling function $\lceil \_ \rceil$ applied to $\kappa_1(1 + \kappa_3)$ is just to make sure that the use is not δ.

3.4.3 Let-expressions

The following rule for let-expressions best illustrates how the evaluation order is taken into account in our type system:

$$\frac{\Gamma_1 \vdash M_1 : \lceil \tau_1 \rceil \qquad \Gamma_2, x : \lceil \tau_1 \rceil \vdash M_2 : \tau}{\Gamma_1; \Gamma_2 \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau} \qquad \text{(T-LET)}$$

Here, the premise means that heap values are accessed by M_1 as described by Γ_1 and by M_2 as described by $\Gamma_2, x : \lceil \tau_1 \rceil$. In ordinary linear type systems, the total use of heap values in $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ was computed by adding Γ_1 and Γ_2. However, since M_2 is evaluated only after M_1 has been fully evaluated, we estimate the total access of heap values by *sequential composition* $\Gamma_1; \Gamma_2$. We require the type of M_1

not to contain δ at top-level (by applying $\lceil\_\rceil$ to τ_1), in order to make sure that every δ-value represented by Γ_1 is really "used up" during the evaluation of M_1 and does not escape to the rest of computation.

3.4.4 Access to heap values

The following is the rule for closure invocation:

$$\frac{\Gamma, x:\tau_1 \vdash M:\tau_2 \qquad \kappa > 0}{(f:\tau_3 \to^{\kappa,1} \tau_1 + V:\tau_3); \Gamma \vdash \mathbf{let}\ x = \mathit{apply}^{\kappa}(f,V)\ \mathbf{in}\ M:\tau_2} \text{(T-App)}$$

The access to heap values by $\mathit{apply}^{\kappa}(f,V)$ is estimated as $f:\tau_3 \to^{\kappa,1} \tau_1 + V:\tau_3$. Note that the first use κ of the type of f should not be zero, and that the second use is 1, since the function is used once here. The access of heap values except for x by M is estimated as Γ. Since $\mathit{apply}^{\kappa}(f,V)$ is evaluated *before* M is evaluated, the total access can be estimated as their sequential composition $(f:\tau_3 \to^{\kappa,1} \tau_1 + V:\tau_3); \Gamma$.

The rule for reading the first element of a pair is:

$$\frac{\Gamma, y:\tau_1', x:\tau_1 \times^{\kappa'} \tau_2 \vdash M:\tau_3 \qquad \kappa > 0}{\Gamma, x:(\tau_1 + \tau_1') \times^{\kappa;\kappa'} \tau_2 \vdash \mathbf{let}\ y = \mathbf{fst}^{\kappa}(x)\ \mathbf{in}\ M:\tau_3} \text{(T-Fst)}$$

The premise implies that the pair is used as a κ'-value in M *after* being used as a κ-value in $\mathbf{fst}^{\kappa}(x)$. So, the total use of x is estimated as $\kappa;\kappa'$. The first element may be accessed through either y or x, hence the total use of the first element is estimated as $\tau_1 + \tau_1'$.

Similarly, the rule for extraction of the second element of the pair is:

$$\frac{\Gamma, y:\tau_2', x:\tau_1 \times^{\kappa'} \tau_2 \vdash M:\tau_3 \qquad \kappa > 0}{\Gamma, x:\tau_1 \times^{\kappa;\kappa'} (\tau_2 + \tau_2') \vdash \mathbf{let}\ y = \mathbf{snd}^{\kappa}(x)\ \mathbf{in}\ M:\tau_3} \text{(T-Snd)}$$

3.4.5 Rules for conditionals

In the following rule for conditional expressions, we require that the then-part and the else-part has the same typing.

$$\frac{\Gamma \vdash M_1:\tau_1 \qquad \Gamma \vdash M_2:\tau_1}{V:\mathit{int} + \Gamma \vdash \mathbf{if0}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2:\tau_1} \text{(T-If)}$$

4 Type Soundness

This section sketches a proof of the type soundness property that no invalid heap access occurs during evaluation of well-typed expressions. The full proof is given in the technical report [12].

The type soundness is formally stated as follows:

Theorem 4.1: If $\emptyset \vdash M:\tau$, then $(\{\}, M) \not\rightsquigarrow^{*}$ **Error**.

We want to show this property as usual [8, 22], by defining a typing system for a run-time state (H, M) and proving that the reduction relation preserves typing and that no well-typed state (H, M) causes an error immediately. Unfortunately, however, the reduction relation $\rightsquigarrow$ defined in Section 2.3 is not suitable for this purpose: a key feature of our type system is to capture information on the order of heap access (recall (T-Let), for example), whereas the flat representation of a heap loses that information. Therefore, we define an alternative operational semantics that represents a heap by using nested letrec-expressions, and show the soundness of the type system with respect to that semantics. Although it includes rather strange reduction rules, it is easy to see that the alternative semantics is essentially equivalent to the original semantics.

Before presenting the alternative semantics, we explain why the reduction relation $\rightsquigarrow$ fails to preserve typing. Let us consider the following configuration:

$$(\{w \mapsto^1 \langle 1,2\rangle, x \mapsto^1 \langle w,3\rangle\}, \mathbf{let}\ v = M_1\ \mathbf{in}\ M_2).$$

where $M_1 = \mathbf{let}\ y = \mathbf{fst}^{\delta}(x)\ \mathbf{in}\ \mathbf{let}\ z = \mathbf{fst}^{\delta}(y)\ \mathbf{in}\ z$. Then, $x:(\mathit{int} \times^{\delta} \mathit{int}) \times^{\delta} \mathit{int} \vdash M_1 : \mathit{int}$. So, if $x:(\mathit{int} \times^{1} \mathit{int}) \times^{1} \mathit{int} \vdash M_2 : \mathit{int}$, then by (T-Let),

$$x:(\mathit{int} \times^{1} \mathit{int}) \times^{1} \mathit{int} \vdash \mathbf{let}\ v = M_1\ \mathbf{in}\ M_2 : \mathit{int},$$

since $((\mathit{int} \times^{\delta} \mathit{int}) \times^{\delta} \mathit{int}); ((\mathit{int} \times^{1} \mathit{int}) \times^{1} \mathit{int}) = (\mathit{int} \times^{1} \mathit{int}) \times^{1} \mathit{int}$. So it is fine that w is a linear pair. However, if we reduce the configuration by (R-Fst), the resulting configuration:

$$(\{w \mapsto^1 \langle 1,2\rangle, x \mapsto^1 \langle w,3\rangle\},\ \mathbf{let}\ v = (\mathbf{let}\ z = \mathbf{fst}^{\delta}(w)\ \mathbf{in}\ z)\ \mathbf{in}\ M_2)$$

requires w to be an ω-pair: w is accessed as a linear pair in M (through x) and as a δ-pair in $\mathbf{fst}^{\delta}(w)$, and the type system cannot capture the order between the two uses (because the access is made through different variables x and w).

One way to avoid the above problem would be to extend the type system so that the order of access to different variables can be expressed, as in [11]. Since the resulting typing rules would become rather complex, we instead redefine the operational semantics. The idea is, before losing information on the order of heap access, to split the heap space in advance by taking the order into account. For example, the above configuration is first converted to something like:

$$\mathbf{let}\ v = (\{w \mapsto^{\delta} \langle 1,2\rangle, x \mapsto^{\delta} \langle w,3\rangle\}, M_1)\ \mathbf{in}\ (\{w \mapsto^1 \langle 1,2\rangle, x \mapsto^1 \langle w,3\rangle\}, M_2).$$

Since the heap space has been split for the definition part and for the body part of the let-expression, we can safely throw away information that the definition part is evaluated before the body part. So, it can be reduced to:

$$\mathbf{let}\ v = (\{w \mapsto^{\delta} \langle 1,2\rangle, x \mapsto^{\delta} \langle w,3\rangle\}, \mathbf{let}\ z = \mathbf{fst}^{\delta}(w)\ \mathbf{in}\ z)\ \mathbf{in}\ (\{w \mapsto^1 \langle 1,2\rangle, x \mapsto^1 \langle w,3\rangle\}, M_2).$$

In order to express the above kind of *nested* heaps and expressions, we introduce a new class of expressions called *dynamic expressions* and define a reduction relation for dynamic expressions. Note that this new operational semantics is introduced just for proving Theorem 4.1. The actual implementation can be based on the semantics defined in Section 2.3 and no cost for splitting heap needs to be paid.

4.1 Dynamic expressions

The syntax of expressions given in Section 2 is extended to that of dynamic expressions as follows:

Definition of $\kappa_1 - \kappa_2$

$\kappa_1 \backslash \kappa_2$	0	δ	1	ω
0	0	undef	undef	undef
δ	δ	δ	undef	undef
1	1	1	0	undef
ω	ω	ω	ω	ω

Definition of $\kappa_1 + \kappa_2$

$\kappa_1 \backslash \kappa_2$	0	δ	1	ω
0	0	δ	1	ω
δ	δ	ω	ω	ω
1	1	ω	ω	ω
ω	ω	ω	ω	ω

Definition of $\kappa_1 ; \kappa_2$

$\kappa_1 \backslash \kappa_2$	0	δ	1	ω
0	0	δ	1	ω
δ	δ	δ	1	ω
1	1	ω	ω	ω
ω	ω	ω	ω	ω

Definition of $\kappa_1 \cdot \kappa_2$

$\kappa_1 \backslash \kappa_2$	0	δ	1	ω
0	0	0	0	0
δ	0	δ	δ	ω
1	0	δ	1	ω
ω	0	ω	ω	ω

Definition of $\lceil \kappa \rceil$

κ	0	δ	1	ω
$\lceil \kappa \rceil$	0	1	1	ω

Definition of $\lfloor \kappa \rfloor$

κ	0	δ	1	ω
$\lfloor \kappa \rfloor$	0	0	1	ω

Figure 3: Operations on uses

Relation $\leq$

$$
\begin{array}{l}
\mathit{int} \leq \mathit{int} \\
(\tau_1 \times^{\kappa} \tau_2) \leq (\tau_1' \times^{\kappa'} \tau_2') \text{ if } (\tau_1 \leq \tau_1' \wedge \tau_2 \leq \tau_2' \wedge \kappa \leq \kappa') \\
(\tau_1 \to^{\kappa_1,\kappa_2} \tau_2) \leq (\tau_1 \to^{\kappa_1',\kappa_2'} \tau_2) \text{ if } (\kappa_1 \leq \kappa_1' \wedge \kappa_2 \leq \kappa_2') \\
\Gamma \leq \Gamma' \text{ if } \mathit{dom}(\Gamma) \subseteq \mathit{dom}(\Gamma') \text{ and } \Gamma(x) \leq \Gamma'(x) \text{ for each } x \in \mathit{dom}(\Gamma).
\end{array}
$$

Binary operations ($\mathbf{op} = +, ;$)

$$
\begin{array}{rcl}
\mathit{int}\mathbf{op}\mathit{int} & = & \mathit{int} \\
(\tau_1 \times^{\kappa} \tau_2)\mathbf{op}(\tau_1' \times^{\kappa'} \tau_2') & = & \begin{cases} (\tau_1 \mathbf{op} \tau_1') \times^{\kappa \mathbf{op} \kappa'} (\tau_2 \mathbf{op} \tau_2') \text{ if } \tau_1 \mathbf{op} \tau_1' \text{ and } \tau_2 \mathbf{op} \tau_2' \text{ are well defined.} \\ \text{undefined} \quad \text{otherwise} \end{cases} \\
(\tau_1 \to^{\kappa_1,\kappa_2} \tau_2)\mathbf{op}(\tau_1 \to^{\kappa_1',\kappa_2'} \tau_2) & = & \tau_1 \to^{\kappa_1 \mathbf{op} \kappa_1', \kappa_2 \mathbf{op} \kappa_2'} \tau_2 \\
(\Gamma_1 \mathbf{op} \Gamma_2)(x) & = & \begin{cases} \Gamma_1(x)\mathbf{op}\Gamma_2(x) & \text{if } x \in \mathit{dom}(\Gamma_1) \cap \mathit{dom}(\Gamma_2) \\ \Gamma_1(x) & \text{if } x \in \mathit{dom}(\Gamma_1) \backslash \mathit{dom}(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in \mathit{dom}(\Gamma_2) \backslash \mathit{dom}(\Gamma_1) \end{cases}
\end{array}
$$

Unary operations ($\mathbf{op} = \lceil \_ \rceil, \kappa \cdot \_$)

$$
\begin{array}{rcl}
\mathbf{op}(\mathit{int}) & = & \mathit{int} \\
\mathbf{op}(\tau_1 \times^{\kappa} \tau_2) & = & \mathbf{op}(\tau_1) \times^{\mathbf{op}(\kappa)} \mathbf{op}(\tau_2) \\
\mathbf{op}(\tau_1 \to^{\kappa_1,\kappa_2} \tau_2) & = & \tau_1 \to^{\mathbf{op}(\kappa_1),\mathbf{op}(\kappa_2)} \tau_2 \\
(\mathbf{op}(\Gamma))(x) & = & \mathbf{op}(\Gamma(x))
\end{array}
$$

Figure 4: Definitions of a relation and operations on types and type environments

$$
\mathbf{let}\ x = h^{\kappa}\ \mathbf{in}\ M \xrightarrow{\epsilon} \mathbf{letrec}\ z = h^{\kappa}\ \mathbf{in}\ [z/x]M\ (z \text{ fresh}) \qquad \text{(DR-Heap)}
$$

$$
\mathbf{let}\ u = \mathbf{fst}^{\kappa}(x)\ \mathbf{in}\ M \xrightarrow{x=\langle V_1,V_2\rangle^{\kappa}} [V_1/u]M \qquad \text{(DR-Fst)}
$$

$$
\frac{D \xrightarrow{x=h^{\kappa'}} D' \qquad \kappa \geq \kappa' > 0}{\mathbf{letrec}\ x = h^{\kappa}\ \mathbf{in}\ D \xrightarrow{\epsilon} \mathbf{letrec}\ x = h^{\kappa-\kappa'}\ \mathbf{in}\ D'} \qquad \text{(DR-Read)}
$$

$$
\frac{D \xrightarrow{x=h^{\kappa'}} D' \qquad (\kappa' > \kappa) \vee (\kappa' = 0)}{\mathbf{letrec}\ x = h^{\kappa}\ \mathbf{in}\ D \xrightarrow{\epsilon} \mathbf{Error}} \qquad \text{(DR-Error)}
$$

$$
\mathbf{letrec}\ x = h^{\kappa_1;\kappa_2}\ \mathbf{in}\ \mathbf{let}\ y = D_1\ \mathbf{in}\ D_2 \xrightarrow{x} \mathbf{let}\ y = \mathbf{letrec}\ x = h^{\kappa_1}\ \mathbf{in}\ D_1\ \mathbf{in}\ \mathbf{letrec}\ x = h^{\kappa_2}\ \mathbf{in}\ D_2 \qquad \text{(DR-Split)}
$$

$$
\begin{array}{l}
\mathbf{let}\ x = (\mathbf{letrec}\ \vec{z} = \vec{h}^{\vec{\kappa_1}}\ \mathbf{in}\ \mathbf{letrec}\ \vec{w} = \vec{h'}^{\vec{\kappa_3}}\ \mathbf{in}\ V)\ \mathbf{in}\ (\mathbf{letrec}\ \vec{z} = \vec{h}^{\vec{\kappa_2}}\ \mathbf{in}\ M) \\
\xrightarrow{\epsilon} \mathbf{letrec}\ \vec{z} = \vec{h}^{\lfloor \vec{\kappa_1} \rfloor;\vec{\kappa_2}}\ \mathbf{in}\ \mathbf{letrec}\ \vec{w} = \vec{h'}^{\lfloor \vec{\kappa_3} \rfloor}\ \mathbf{in}\ [V/x]M
\end{array} \qquad \text{(DR-Var)}
$$

Figure 5: Main reduction rules for dynamic expressions

Definition 4.2 [dynamic expressions]: The set $\mathcal{D}$ of dynamic expressions, ranged over by D, is given by the following syntax:

$$D \ ::= \ M \mid \textbf{let } x = D_1 \textbf{ in } D_2 \mid \textbf{letrec } x = h^\kappa \textbf{ in } D$$

Here, we introduced a letrec-expression $\textbf{letrec } x = h^\kappa \textbf{ in } D$ to express a heap binding, and extended the syntax of let-expressions so that heap bindings and let-bindings can be nested. By using the new syntax, the nested heap and expression

$$\textbf{let } v = (\{x \mapsto^\delta \langle w, 3\rangle\}, M_1) \textbf{ in } (\{x \mapsto^1 \langle w, 3\rangle\}, M_2)$$

can be expressed by the following dynamic expression:

$$\begin{array}{l}\textbf{let } v = (\textbf{letrec } x = \langle w, 3\rangle^\delta \textbf{ in } M_1)\\ \textbf{in letrec } x = \langle w, 3\rangle^1 \textbf{ in } M_2\end{array}$$

The typing rules for expressions can be easily extended for dynamic expressions. The new rules are as follows:

$$\frac{\Gamma, x : \tau_1 \times^\kappa \tau_2 \vdash D : \tau_3 \qquad x \notin \{V_1, V_2\}}{\Gamma + V_1 : \tau_1 + V_2 : \tau_2 \vdash \textbf{letrec } x = \langle V_1, V_2\rangle^\kappa \textbf{ in } D : \tau_3} \ \text{(DT-PAIR)}$$

$$\frac{\begin{array}{c}\Gamma_1, x : \tau_1, f : \tau_1 \to^{\kappa_1,\kappa_2} \lceil\tau_2\rceil \vdash M_1 : \tau_2\\ \Gamma_2, f : \tau_1 \to^{\kappa_3,\kappa_4} \tau_2 \vdash D : \tau_3\\ \kappa \geq \kappa_3(1+\kappa_1)\end{array}}{\kappa_4(1+\kappa_2)\lceil\Gamma_1\rceil + \Gamma_2 \vdash \textbf{letrec } f = \lambda^\kappa x.M_1 \textbf{ in } D : \tau_3} \ \text{(DT-ABS)}$$

4.2 Operational semantics of dynamic expressions

We define an operational semantics by using a reduction relation $D \xrightarrow{l} E$. D is a dynamic expression and E is either a dynamic expression or a special constant **Error** representing invalid heap access. The label l shows which heap value is accessed in the reduction step. It is either ϵ, which indicates that an internal heap value is accessed, or $x = h^\kappa$, which indicates that the heap address x is accessed as a value with use κ, or x, which indicates that the heap of address x is split as explained above.

We show only the key rules in Figure 5. The rule (DR-HEAP) corresponds to the rule (R-HEAP) of the original semantics. The rules (DR-FST), (DR-READ) and (DR-ERROR) correspond to the rules (R-FST) and (F-FST-ERROR). The rule (DR-SPLIT) is the most important rule: it allows the current heap to be split into that for the definition part of a let-expression and that for the body part. The rule (DR-VAR) corresponds to the rule (R-VAR), but it allows split heap bindings to be merged after the definition part of a let-expression has been fully evaluated. In the rule, $\textbf{letrec } \vec{z} = \vec{h}^{\vec{\kappa}} \textbf{ in } D$ is a shorthand for $\textbf{letrec } z_1 = h_1^{\kappa_1} \textbf{ in } \cdots \textbf{letrec } z_n = h_n^{\kappa_n} \textbf{ in } D$.

Example 4.3: The expression in Example 2.3 is reduced as follows:

$$\begin{array}{ll} & \textbf{let } x = \langle 1,2\rangle^1 \textbf{ in}\\ & \quad \textbf{let } y = \textbf{fst}^\delta(x) \textbf{ in let } z = \textbf{snd}^1(x) \textbf{ in } z\\ \xrightarrow{\epsilon} & \textbf{letrec } x = \langle 1,2\rangle^1 \textbf{ in}\\ & \quad \textbf{let } y = \textbf{fst}^\delta(x) \textbf{ in let } z = \textbf{snd}^1(x) \textbf{ in } z\\ \xrightarrow{x} & \textbf{let } y = (\textbf{letrec } x = \langle 1,2\rangle^\delta \textbf{ in fst}^\delta(x)) \textbf{ in}\\ & \quad (\textbf{letrec } x = \langle 1,2\rangle^1 \textbf{ in let } z = \textbf{snd}^1(x) \textbf{ in } z)\\ \xrightarrow{\epsilon} & \textbf{let } y = (\textbf{letrec } x = \langle 1,2\rangle^\delta \textbf{ in } 1) \textbf{ in}\\ & \quad (\textbf{letrec } x = \langle 1,2\rangle^1 \textbf{ in let } z = \textbf{snd}^1(x) \textbf{ in } z)\\ \xrightarrow{\epsilon} & \textbf{letrec } x = \langle 1,2\rangle^{\delta;1} \textbf{ in let } z = \textbf{snd}^1(x) \textbf{ in } z\\ \xrightarrow{x} & \textbf{let } z = (\textbf{letrec } x = \langle 1,2\rangle^1 \textbf{ in snd}^1(x)) \textbf{ in}\\ & \quad (\textbf{letrec } x = \langle 1,2\rangle^0 \textbf{ in } z)\\ \xrightarrow{\epsilon} & \textbf{let } z = (\textbf{letrec } x = \langle 1,2\rangle^0 \textbf{ in } 2) \textbf{ in}\\ & \quad (\textbf{letrec } x = \langle 1,2\rangle^0 \textbf{ in } z)\\ \xrightarrow{\epsilon} & \textbf{letrec } x = \langle 1,2\rangle^0 \textbf{ in } 2\end{array}$$

4.3 Proof sketch of Theorem 4.1

Theorem 4.1 follows immediately from the soundness of the type system with respect to the new operational semantics and the fact that if an expression is reduced to **Error** in the original semantics then it is also the case in the new semantics.

The type soundness with respect to the new operational semantics is stated as the two lemmas given below. The subject reduction property (Lemma 4.5) is a little more complicated than the usual one because even a well-typed dynamic expression may be reduced to an ill-typed expression if a heap value is split in an inappropriate way. The second statement of the lemma says that, if heap splitting (the rule (DR-SPLIT)) can be applied to a well-typed expression, there is always a *good* splitting that preserves the well-typeness of the expression.

Lemma 4.4 [Lack of Immediate Error]: If $\Gamma \vdash D : \tau$, then $D \not\xrightarrow{\epsilon}$ **Error**.

Lemma 4.5 [Subject Reduction]:

- If $\Gamma \vdash D : \tau$ and $D \xrightarrow{\epsilon} D'$, then $\Gamma \vdash D' : \tau$; and
- If $\Gamma \vdash D : \tau$ and $D \xrightarrow{x} D'$, then there exists D'' such that $D \xrightarrow{x} D''$ and $\Gamma \vdash D'' : \tau$.

We write $\xrightarrow{\hat{\epsilon}}$ for the relation $\xrightarrow{x_1} \cdots \xrightarrow{x_n}\xrightarrow{\epsilon}$. We also write $D \uparrow$ if D is reduced to **Error** no matter how the heap bindings in D are split, i.e., if $D \xrightarrow{\hat{\epsilon}}$ **Error** and there is no $D' \in \mathcal{D}$ such that $D \xrightarrow{\hat{\epsilon}} D'$. The correspondence between the new semantics and the original semantics can be stated as follows:

Lemma 4.6: There is a relation $\mathcal{R}(\subseteq (\mathcal{H} \times \mathcal{E}) \times \mathcal{D})$ that satisfies the following conditions:

- $(\{\}, M)\mathcal{R}M$;
- if $(H, M) \rightsquigarrow (H', M')$ and $(H, M)\mathcal{R}D$, then either $D \xrightarrow{\hat{\epsilon}} D'$ and $(H', M')\mathcal{R}D'$ for some D' or $D \uparrow$; and
- if $(H, M) \rightsquigarrow$ **Error** and $(H, M)\mathcal{R}D$, then $D \uparrow$.

5 Type Reconstruction

Use annotations can be automatically inferred from unannotated expressions. Since it is done basically in the same way as that for previous linear type systems [9, 22], we explain it only informally. The basic idea is to introduce variables ranging over uses and types, and constraints on them, so

that we can express the most general typing (principal typing) of an expression. A new type judgment is of the form $\Gamma, C \vdash M : \tau$, where a set of constraints C specifies which set of uses/types each use/type variable (in Γ, M, or τ) can range over. For instance, **let** $x = \langle y, y\rangle$ **in** x is typed as: $y : \alpha, \{\alpha \geq \beta + \gamma, i \neq \delta, i \geq j\} \vdash \mathbf{let}\ x = \langle y, y\rangle^i\ \mathbf{in}\ x : \beta \times^j \gamma$. This typing is *principal* in the sense that all the typings derivable from the rules in Section 3 can be obtained by instantiating variables α, β, γ, i and j so that the constraint is satisfied. (We do not give the formal definition of principal typings here; it is basically the same as that in [22].)

Given an unannotated expression M, the inference of a use annotation proceeds as follows (actually, the second and third steps may overlap):

1. Attach a fresh use variable to each place where use annotation is required (let the annotated expression be M').
2. Compute a principal typing $\Gamma, C \vdash M' : \tau$.
3. Solve the constraint C and apply the obtained substitution to M'.

In the following subsections, we explain the second and third steps in a little more detail, and then discuss the cost of the analysis.

5.1 Computing a principal typing

An algorithm for computing a principal typing is obtained by constructing syntax-directed typing rules for the new type judgment form and by reading them from bottom to up. For example, we can merge the rules (T-Pair) and (T-Weak) and obtain the following rule for the new type judgment form:

$$\frac{\Gamma, x : \tau_1 \times^\kappa \tau_2, C \vdash M : \tau_3 \quad C \models \Gamma' \geq (\Gamma + V_1 : \tau_1 + V_2 : \tau_2) \quad C \models \kappa \neq \delta}{\Gamma', C \vdash \mathbf{let}\ x = \langle V_1, V_2\rangle^\kappa\ \mathbf{in}\ M : \tau_3} \quad \text{(TR-Pair)}$$

Here, $C \models \Gamma_1 \geq \Gamma_2$ means that, for any substitution θ on type/use variables, if θC is satisfied then $\theta\Gamma_1 \geq \theta\Gamma_2$ is also satisfied. The above rule says that in order to compute a typing for **let** $x = \langle V_1, V_2\rangle^\kappa$ **in** M, we should first compute a typing for M and then add new constraints entailing $\Gamma' \geq \Gamma + V_1 : \tau_1 + V_2 : \tau_2$ and $\kappa \neq \delta$.

One major difference from the previous linear type inference [8, 22] is that constraints on type variables appear in C (recall the typing for **let** $x = \langle y, y\rangle$ **in** x given above). They are of the form $\tau_1 \geq \tau_2$ or $\tau_1 \sim \tau_2$ (meaning τ_1 and τ_2 are *compatible* in the sense that $\tau_1 + \tau_2$ and $\tau_1; \tau_2$ are well defined). Here, the righthand type expression τ_2 of $\tau_1 \geq \tau_2$ may contain $+$, $\lceil\_\rceil$, etc. as constructors of type expressions (rather than as functions on types).

5.2 Solving constraints on types and uses

An algorithm for solving constraints can be divided into two steps. First, a set of constraints on type/use variables can be simplified into a pair of a set of constraints on use variables of the form $i \geq \kappa$[2] and a set of "trivial" constraints on type variables of the form $\alpha \sim \beta$ or $\alpha \geq \tau$ where τ is a type expression (such as $\beta + \gamma$) constructed only from type variables. This simplification is performed by *partial* unification of types, with some uses being kept different. For example, given a constraint $\alpha \geq (\beta \times^j \mathit{real}^i) + \gamma$, we can instantiate α and γ with $\beta' \times^{j'} \mathit{real}^{i'}$ and $\beta'' \times^{j''} \mathit{real}^{i''}$ for fresh variables $\beta', \beta'', i', i'', j', j''$ and reduce the constraint to $\{\beta' \geq \beta + \beta'', i' \geq i + i'', j' \geq j + j''\}$.

Next, since the set of trivial constraints on type variables always has a solution (let all the remaining type variables be instantiated with, say, *int*), we can obtain the least assignment to use variables by solving the set of constraints on use variables. We can use the simple iterative method used in previous linear type systems [8, 9]. Of course, the least assignment to some use variables cannot be completely determined until the whole program is given. For example, given an expression of type real^i, we cannot find the least assignment to the use variable i until we know all the places where the evaluation result of the expression is used. For these unknown variables, we can either simply assign ω or delay constraint solving.

One important difference from the previous linear type inference is that the least solution is not necessarily the best solution from the viewpoint of memory management. For example, consider an expression **let** $y = \langle 1, 2\rangle$ **in** (**let** $x = \mathbf{fst}(y)$ **in** x). The least annotation is **let** $y = \langle 1, 2\rangle^1$ **in** (**let** $x = \mathbf{fst}^\delta(y)$ **in** x). In this expression, y cannot be deallocated since the only access to the pair y ($\mathbf{fst}(y)$) is annotated with δ. Therefore, it is better to annotate the expression as **let** $y = \langle 1, 2\rangle^1$ **in** (**let** $x = \mathbf{fst}^1(y)$ **in** x). We can think of two ways for dealing with this. One way is to classify uses into those for annotating heap allocation and those for annotating heap access. After the least solution is obtained, we can maximize heap access annotations as long as no heap allocation annotations increase. The other way is to extend a target language with an explicit deallocation operation *free*(Γ): It deallocates heap values as if the heap were accessed as described by Γ. For example, $\mathit{free}(x : \mathit{real}^1, y : \mathit{real}^\delta)$ deallocates x but not y. The rule (T-Weak) can be changed as follows ($\Gamma - \Gamma'$ is the least type environment Γ'' such that $\Gamma'; \Gamma'' = \Gamma$):

$$\frac{\Gamma' \vdash M : \tau \qquad \Gamma' \leq \Gamma}{\Gamma \vdash M; \mathit{free}(\Gamma - \Gamma') : \tau} \quad \text{(T-Weak')}$$

This change avoids forgetting to deallocate linear values. We can optimize the result by moving the operation *free* leftward and merging it with heap access annotation as far as possible. Although there is no guarantee that these methods give the *best* use annotation, we expect that both give enough good an annotation in practice.

5.3 Cost of the analysis

Unfortunately, the computational cost for type reconstruction can be exponential in the size of an input in the worst case. The reason is that the number of use variables can be linear in the size of the type of a variable appearing in an input expression and that the size of a type can be exponential in the size of an input expression. For example, if a variable x is required to have type $(\mathit{int} \times \mathit{int}) \times (\mathit{int} \times \mathit{int})$ by the context, its typing is:

$$x : (\mathit{int} \times^{i_1} \mathit{int}) \times^{i_3} (\mathit{int} \times^{i_2} \mathit{int}), \{i_1 \geq i_1', i_2 \geq i_2', i_3 \geq i_3'\} \vdash x : (\mathit{int} \times^{i_1'} \mathit{int}) \times^{i_3'} (\mathit{int} \times^{i_2'} \mathit{int}).$$

[2] Note that a constraint $i \neq \delta$ can be expressed as $i \geq \lceil i \rceil$.

So, the number of use variables is linear in the size 3 of the type $(int \times int) \times (int \times int)$. Next, consider an expression

$$\textbf{let } x_0 = 1 \textbf{ in let } x_1 = \langle x_0, x_0 \rangle \textbf{ in let } x_2 = \langle x_1, x_1 \rangle \textbf{ in} \\ \cdots \textbf{let } x_n = \langle x_{n-1}, x_{n-1} \rangle \textbf{ in } x_n.$$

The size of the type of the expression is exponential in n.

In spite of the above fact, we expect that the reconstruction can be performed efficiently for realistic programs for the following reasons. First, the above problem occurs only when tuple or record types are extraordinarily nested; a huge flat tuple type expression $int \times int \times \cdots \times int$ or an arrow type expression $\tau_1 \to \tau_2$ do not cause such problems. Second, if the computational cost is really problematic for some program, we can save the cost by sacrificing the accuracy of the analysis. For example, we can restrict the + operator so that $(\tau_1 \times^{\kappa} \tau_2) + (\tau_1' \times^{\kappa'} \tau_2')$ is defined only if $\tau_1 = \tau_1'$ and $\tau_2 = \tau_2'$. This forces many type expressions to be shared and saves the time and space of the analysis (in fact, the previous analysis in [8, 9], which imposes a similar restriction, is performed in polynomial time for the monomorphic type system).

6 Extensions

We have so far considered a simply-typed λ-calculus with recursion and pairs. It can be extended by introducing polymorphism on uses and types, and by adding other kinds of data such as recursive data structures and reference cells. For lack of space, we only briefly discuss how to introduce polymorphism, constants such as real numbers, and recursive data structures.

6.1 Polymorphism

Because the type system presented in Section 3 is monomorphic in both uses and types, the analysis of (quasi-)linear values is often rough. Consider the following expression:

$$\textbf{let } f = \lambda x.\langle x, x \rangle \textbf{ in let } z = apply(f, 2) \textbf{ in} \\ \textbf{let } w = apply(f, 3) \textbf{ in } M.$$

If either z or w is used as an ω-value in M, the other is also forced to be an ω-value, because the code for the pair creation (i.e., the body of the function f) is shared.

In order to avoid the above problem, we can introduce polymorphism on uses. Then, f can be given a type $\forall i \neq \delta.(int \to^{\omega,\omega} (int \times^{i} int))$ and the above expression can be annotated, for example, as

$$\textbf{let } f = \Lambda i.\lambda x.\langle x, x \rangle^{i} \textbf{ in let } z = apply(f[1], 2) \textbf{ in} \\ \textbf{let } w = apply(f[\omega], 3) \textbf{ in } M.$$

Here, uses are explicitly passed as parameters to polymorphic functions.

In general, we need to introduce a constrained type scheme of the form $\forall \alpha_1, \ldots, \alpha_n, i_1, \ldots, i_k :: C.\tau$, where C is a set of constraints on type and use variables such as $\alpha_1 \geq \alpha_2 + \alpha_3$ and $i_1 \geq i_2 + 1$. A new rule for polymorphic let-expressions roughly looks like:

$$\frac{\begin{array}{c}\Gamma_1, C_1 \wedge C_2 \vdash M_1 : \lceil \tau_1 \rceil \\ \Gamma_2, x : \forall \vec{\alpha}, \vec{i} :: C_2.\lceil \tau_1 \rceil, C_3 \vdash M_2 : \tau_2 \\ \vec{\alpha}, \vec{i} \text{ do not affect the values of variables in } \Gamma_1, M\end{array}}{\Gamma_1; \Gamma_2, C_1 \wedge C_3 \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau_2} \quad \text{(T-LetPoly)}$$

6.2 Large constants

So far, we have considered only integers as constants. In order to deal with large constants (i.e., constants that cannot be represented in one word and must be allocated in a heap), we need to annotate each allocation of a large constant. For example, allocation of a linear real number 2.0 is annotated as $\textbf{let } x = 2.0^{1} \textbf{ in } \cdots$. The type of a large constant is also annotated with a use. For instance, the type of linear real numbers is represented by $real^{1}$. The typing rule for real numbers is given as follows:

$$\frac{\Gamma, x : real^{\kappa} \vdash M : \tau}{\Gamma \vdash \textbf{let } x = r^{\kappa} \textbf{ in } M : \tau}$$

6.3 Primitive functions

Primitive functions can be treated as constants of polymorphic types, and each occurrence of a primitive function can be annotated with uses. For example, the primitive + on real numbers can be assigned a type

$$\forall i_1, i_2, i_3, i_4 :: \{i_1 \geq \delta, i_2 \geq \delta, i_3 \geq \delta, i_4 \neq \delta\}. \\ ((real^{i_1} \times^{i_3} real^{i_2}) \to^{\omega,\omega} real^{i_4}).$$

The expression $\textbf{let } x = y + z \textbf{ in } \cdots$ can be annotated like: $\textbf{let } x = +[1,1,1,1]\langle y, z \rangle^{1} \textbf{ in } \cdots$.

6.4 Recursive data structures

Recursive data structures such as lists and trees can be treated in the same way as pairs, by annotating their type constructors with uses. The type of a list of values of type τ is expressed as $\tau\ list^{\kappa}$, where κ expresses the use of each cons cell of the list. For example, $real^{1}\ list^{\omega}$ is the type of a list whose cons cells may be accessed many times in an arbitrary manner, but whose elements are real numbers that can be accessed only linearly. Primitives for lists can be given the following types:

$$\begin{array}{lll} :: & : & \forall \alpha, i.((\alpha \times^{i} \alpha\ list^{i}) \to^{\omega,\omega} \alpha\ list^{i}) \\ nil & : & \forall \alpha, i.\alpha\ list^{i} \\ hd & : & \forall \alpha, i :: \{i \geq \delta\}.(\alpha\ list^{i} \to^{\omega,\omega} \alpha) \\ \cdots & & \end{array}$$

In general, we can generate the types of constructors and destructors from datatype declarations of Standard ML [15].

7 Preliminary Experiments

We have implemented a prototype type reconstruction system for the core-ML (i.e., SML [15] without modules) based on our quasi-linear type system and a simple profiler that executes the output program and counts the number of allocated heap values. We first describe the current status of the system in Section 7.1 and then report the results of simple experiments in Section 7.2. The prototype system will be available from `http://www.yl.is.s.u-tokyo.ac.jp/~koba/research/qlinear/`.

7.1 A prototype type reconstruction system

Following the extensions discussed in Section 6, we have implemented a prototype analyzer. It takes an expression of the core-ML as an input, performs type (and use) reconstruction and produces a use-annotated expression. It has

been implemented by using the ML kit version 1 [4] as a front end, and supports full features of the core-ML: records, datatype declarations, references, and exceptions. Polymorphism on types and uses is based on the let-polymorphism discussed in Section 6.

Polymorphic recursion on uses would be sound (just as polymorphic recursion on regions is sound in the region inference [20]), but it is not supported currently. That is because the algorithm would become complex and also because the polymorphic recursion does not seem so crucial for our analysis.

The current system has the following limitations. It produces the least use annotation, not the best one (see Section 5); The analysis of the use of a reference cell is rough; Unnecessary uses are passed as parameters to use-polymorphic functions (they can be removed in the same manner as unnecessary region parameters are removed in a post-path of the region inference [5]); The current system is very slow for several known reasons (it is just because we preferred rapid prototyping and the system will be optimized in the future); A compiler that utilizes the analyzed information for in-place update, etc. has not been implemented yet.

7.2 Results of preliminary experiments

Table 1 shows the result of experiments. The columns 'zero,' 'linear,' and 'omega' respectively show how many heap values of use 0, 1, and ω are allocated dynamically in each program. The rightmost column shows what percentage of heap values are zero or linear. For a use-polymorphic function like $f : \forall i.(real^i \rightarrow^{\omega,\omega} real^i)$, the allocation was counted only once; in other words, creations of instances like $f[1]$ and $f[\omega]$ are not counted as the heap allocation (this is because the current analyzer infers not the use of each instance of a use-polymorphic function but the total use of all the instances).

"sumlist10000" is a program that makes a list of integers from 0 to 10,000 and computes the sum of them. "qsort20" and "msort20" sort a list of 20 real numbers by using the quick sort and merge sort algorithms. They were written in a naive way; the main part of the quick sort program is:

```
fun quick ([]) = []
  | quick (x::l) =
        let val (l1,l2)=divide(x,l)
        in append(quick(l1), x::quick(l2)) end
```

"sieve2000" finds prime numbers less than 2,000 by using the sieve of Eratosthenes. "life" computes the first 10 generations of lives generated from the initial 44 lives. "dangle" and "reynolds3" are programs taken from [5] (with some parameters being changed). "reynolds3u" is a slightly modified version of "reynolds3," in which one function is uncurried so that our analysis works better.

In the cases of sumlist, merge/quick sorts and the sieve of Eratosthenes, almost all of the heap values are linear: non-linear values were only top-level functions. Moreover, by looking at the produced use-annotated expression, we can find that the cons cells generated in merge/quick sorts and the sieve of Eratosthenes can all be updated in-place.

In the case of the life (some curried functions in the original program were uncurried: see discussions in Section 9), dangle, and reynolds3, not all heap values are linear, but the number of linear values is still huge enough to greatly reduce the need for garbage collection. In the Knuth-Bendix and boyer programs, the percentage of linear values is relatively smaller. This is probably because unification of first-order terms performed in the program causes sharing of many heap values.

One should, however, note that the percentages shown in the rightmost column do not necessarily indicate how much memory space can be saved by our analysis: compared with memory management using conventional garbage collection, extra memory space is required for representing use-polymorphic functions and its instances, and it is not yet clear how soon linear values can indeed be deallocated. Also, the current analysis is performed on unoptimized programs; other optimizations such as inlining and hoisting may reduce the ratio of linear values. Therefore, we know for sure the real impact of our analysis only after more serious experiments are carried out (for that purpose, some of the future work described in Section 9 must be completed).

8 Related Work

Previous linear type systems To our knowledge, among previous linear type systems [3, 9, 22] that can automatically infer the usage of values, only Barendsen and Smetsers's uniqueness typing [3] takes an evaluation order into account. The effect of taking an evaluation order into account sometimes depends on a programming style, but it is evident at least for the following functions:

```
fun map f [] = []
  | map f (x::l) = f x::map f l
fun diff ([], S2) = []
  | diff (x::S1, S2) =
      if member(x, S2) then diff(S1, S2)
      else x::diff(S1,S2)
fun move (p, dx) = update(p, X, p.X+dx)
```

The function `f` in `map` and the list `S2` in `diff` (which is a function computing the set difference) are used many times, but are judged to be linear in the quasi-linear type system. The function `move` takes a record representing a point object as the first argument and adds `dx` to its `X` field. The point `p` is accessed twice, but it is also judged to be linear.

The uniqueness typing [3] is different from ours in many ways: the evaluation order is call-by-need,[3] their type system does not have "δ-type" of values that must be used up locally, and instead it relies on a separate analysis for analyzing the order of memory access (so, the analysis is not integrated as a use-type system, and it is rather complex).

Guzmán and Hudak [7] and Odersky [18] also proposed a kind of an extension of a linear type system, which can take an evaluation order into account and check that destructive operations on arrays or lists are safely used. Unlike ours, their approach is to let programmers explicitly declare where destructive operations should be performed (by using special primitives for destructive update of data structures) and check whether they are safe by performing type inference.

Region inference There is an alternative approach to static memory management: region inference [5, 20]. The region inference abstracts a bunch of memory addresses as a region, and estimates its life-time by performing a kind of type inference. Our type system and the region inference have both

[3] Recently, they dealt also with strict let expressions, but the analysis for them seems to be more naive than our analysis [19].

	zero	linear	omega	(zero+linear)/total
sumlist10000	0	40,001	2	100.0%
qsort20	0	448	4	99.1%
msort20	0	496	4	99.2%
sieve2000	0	256,455	8	100.0%
life	0	2,353,116	26,600	98.9%
Knuth-Bendix	0	10,339,410	2,949,292	77.8%
boyer	0	1,163,786	342,642	77.3%
mandelbrot	0	5,672,170	920,944	86.0%
dangle	20,000	20,032	34	99.9%
reynolds3	0	21,515	2,059	91.3%
reynolds3u	0	21,514	13	99.9%

Table 1: The number of allocated heap values

advantages and disadvantages. Because the life-time of data is estimated region-wise, it is difficult to use the region inference for in-place update. Another shortcoming of the region inference is that too many data are often merged into the same region (especially when recursive data structures, higher-order functions, and reference cells are used), and as a result, the analysis of the life-time of data sometimes becomes rough. For example, consider how to represent the type of a list. Because a list may contain an arbitrary number of cons cells, it is impossible to use a distinct region for each cons cell; so, the type of a list must be something like $(\tau\ \mathit{list}, r)$ where r is the region in which all the cons cells are stored. This implies that a cons cell of a list can be deallocated only after all the cons cells of the list become garbage. On the other hand, in our analysis, the type of a list is of the form $\tau\ \mathit{list}^{\kappa}$; if κ is 1, each cons cell can be deallocated separately when it is accessed as a value of use 1. Thus, unlike in the region inference, the life-times of cons cells are not merged. (We do not intend to say that our analysis is always better for lists than the region inference. Indeed, if some cons cell of a list is accessed as an ω-value, our analysis assigns type $\tau\ \mathit{list}^{\omega}$ to the list; hence no cons cells of the list can be deallocated automatically. In this case, the region inference would be much better.)

On the other hand, the advantages of the region inference are that it is completely independent of how often values are accessed, and that the deallocation of data in the same region can be performed in a constant time. So, it may be interesting to use our type system and the region inference as complementary methods for static memory management.

The goal of our analysis is also similar to that of the storage mode analysis [5], which was proposed as a complementary to the region inference. It analyzes whether each access to a value is the last access to the region of the value, and if so, it inserts a code to deallocate all values in the region. Thus, unlike our analysis, with the storage mode analysis, a value can be deallocated only after all the data in the region become garbage.

9 Conclusion and Future Work

We proposed an extended linear type system that can elegantly take an evaluation order into account, and proved its correctness. Static memory management like the one proposed here and the region inference is currently less popular than conventional garbage collection. However, we think it will become very important in the near future. First, static memory management is especially attractive in parallel/distributed environments, since it requires much less communications/synchronizations than conventional garbage collection. Second, with the increasing importance of cache memory, saving the memory space for a program is also likely to save its execution time.

The type system proposed in this paper is for call-by-value languages; it is not yet clear whether a similar type system can be developed for lazy functional languages.

Much work is left to be done to apply our type system to a compiler of ML and know its real impact. We need to work at least on the following issues (although they seem to be fairly straightforward except for the last issue):

Refinement of the type inference system As discussed in Section 5.2, the least use annotation may not be *optimal* from the viewpoint of memory management; so, we need to modify the analyzer so that it can produce a better use annotation.

Transformation for in-place updates We need to implement a compilation path that finds places where deallocation of a heap value is immediately followed by allocation of another value of the same type and replaces it with in-place update.

Combination with conventional garbage collection Since the linear-type-based memory management can automatically deallocate only quasi-linear values, we need a conventional garbage collector as well. There is, however, a subtle problem: since the linear-type-based memory management is not based on the pointer traversal information, it may create a dangling pointer (just as the region-based memory managements does [20]). For example, when the closure $(\lambda x.(\mathbf{fst}(y)+x), \{y = \langle 1.0, 2.0\rangle\})$ is traced at garbage collection time, the second element 2.0 of y may have already been deallocated. There are two ways for dealing with this problem. One way is to exclude the use 0 from the type system so that dangling pointers cannot be created. The other way, which we are currently exploring (with Atsushi Igarashi), is to use the idea of tag-free garbage collection [16, 21]: we can utilize use-annotated types for tracing heap values at garbage collection time, instead of conventional types. For example, from the type $\mathit{real}^1 \times^{\omega} \mathit{real}^0$ of y in the above closure, it is known that the second element of y will not be accessed by the closure and hence it need not be traced by a garbage collector.

The following improvement of the accuracy of the analysis is also future work.

Taking the order of access to different variables into account Because the type system cannot deal with the order of access to different variables, the analysis can be rough when variables are aliased. For example, consider an expression **let** $y = x$ **in** $\mathbf{fst}^{\delta}(x) + \mathbf{fst}^{1}(y)$. Since x and y refer to the same heap value, we could regard x as a linear value; however, the current type system judges x to be an ω-value. We can overcome this problem by representing a type environment as a poset of type bindings in order to express the order of access to different variables, as in an earlier version [10] of Kobayashi's type system for deadlock-freedom [11].

Combining our analysis with other analyses Our analysis of δ-values is weak in dealing with curried functions. Consider an expression $(\lambda z.(\mathbf{fst}(z) + 1.0))\langle 2.0, 3.0\rangle$ and its curried version $((\lambda x.\lambda y.(x + 1.0))2.0)3.0$. For the former expression, δ can be assigned to 2.0, while 1 must be assigned to it for the latter one. This problem may not be so serious because the ordinary compiler optimization uncurries functions as far as possible, but it may be interesting to improve the analysis by using more accurate dataflow information like the one obtained by the region inference.

Acknowledgment

We would like to thank Atsushi Igarashi, Martin Odersky, Kenjiro Taura, Mads Tofte, and the anonymous referees for useful discussions and comments.

References

[1] H. G. Baker. Lively linear lisp – look ma, no garbage! *ACM Sigplan Notices*, 27(8):89–98, 1992.

[2] H. G. Baker. A linear logic quicksort. *ACM Sigplan Notices*, 29(2):13–18, 1994.

[3] E. Barendsen and S. Smetsers. Conventional and uniqueness typing in graph rewrite systems. Technical Report CSI-R9328, Computer Science Institute, University of Nijmegen, 1993. An extended abstract appeared in Proc. of FST&TCS 12, Springer LNCS 761, pp.41-51.

[4] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner. The ML Kit (Version 1). Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.

[5] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 171–183, 1996.

[6] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[7] J. G. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 333–343, 1990.

[8] A. Igarashi. Type-based analysis of usage of values for concurrent programming languages. Master's thesis, Department of Information Science, University of Tokyo, 1997.

[9] A. Igarashi and N. Kobayashi. Type-based analysis of usage of communication channels for concurrent programming languages. In *Proceedings of International Static Analysis Symposium (SAS'97)*, Springer LNCS 1302, pages 187–201, 1997.

[10] N. Kobayashi. A Partially Deadlock-free Typed Process Calculus (I) – A Simple System –. Technical Report 96-02, Department of Information Science, University of Tokyo, September 1996.

[11] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998. A preliminary summary appeared in Proceedings of LICS'97, pages 128–139.

[12] N. Kobayashi. Quasi-linear types. Technical Report 98-02, Department of Information Science, University of Tokyo, 1998. Available through `http://www.yl.is.s.u-tokyo.ac.jp/~koba/publications.html`.

[13] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 358–371, January 1996.

[14] I. Mackie. Lilac : A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):1–39, October 1994.

[15] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[16] G. Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1995.

[17] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 66–76, 1995.

[18] M. Odersky. Observers for linear types. In *Proceedings of 4th European Symposium on Programming (ESOP'92)*, Springer LNCS 582, pages 390–407, 1992.

[19] R. Plasmeijer and M. van Eekelen. Concurrent Clean ver.1.3 language report, 1997.

[20] M. Tofte and J.-P. Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 188–201, 1994.

[21] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 1–11, 1994.

[22] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proceedings of Functional Programming Languages and Computer Architecture*, San Diego, California, pages 1–11, 1995.