Region-Based Memory Management for a Dynamically-Typed Language

Akihito Nagata[1], Naoki Kobayashi [2], and Akinori Yonezawa[1]

[1]Dept. of Computer Science, University of Tokyo[**]
{ganat,yonezawa}@yl.is.s.u-tokyo.ac.jp
[2]Dept. of Computer Science, Tokyo Institute of Technology
kobayasi@cs.titech.ac.jp

Abstract. Region-based memory management scheme has been proposed for the programming language ML. In this scheme, a compiler statically estimates the lifetime of each object by performing an extension of type inference (called region inference) and inserts code for memory allocation and deallocation. Advantages of this scheme are that memory objects can be deallocated safely (unlike with manual memory management using malloc/free) and often earlier than with run-time garbage collection. Since the region inference is an extension of the ML type inference, however, it was not clear whether the region-based memory management was applicable to dynamically-typed programming languages like Scheme. In this paper, we show that the region-based memory management can be applied to dynamically-typed languages by combining region inference and Cartwright et al.'s soft type system.

1 Introduction

Tofte et al. [23] proposed a static memory management scheme called *region inference*. In this scheme, heap space is divided into abstract memory spaces called *regions*. Memory is allocated and deallocated region-wise and every object generated at run-time is placed in one of the regions. A compiler statically estimates the lifetime of each region, and statically inserts code for allocating/deallocating regions.

For example, a source program:

$$\textbf{let } x = (1, 2) \textbf{ in } \lambda y.\ \#1\ x \textbf{ end}$$

is translated into

$$\begin{array}{l}\textbf{letregion } \rho_2 \textbf{ in}\\ \textbf{let } x = (1 \textbf{ at } \rho_1, 2 \textbf{ at } \rho_2) \textbf{ at } \rho_3\\ \quad \textbf{in } \lambda y.\ \#1\ x \textbf{ at } \rho_4 \textbf{ end}\\ \textbf{end}\end{array}$$

[**] Nagata's current affiliation: OS Development Dept. R&D Div. Sony Computer Entertainment Inc.

Here, #1 is the primitive for extracting the first element from a pair, and ρ_i stands for a region. **letregion** ρ **in** e **end** is a construct for allocating and deallocating a region. It first creates a new region ρ, and evaluates e. After evaluating e, it deallocates ρ and returns the evaluation result. v **at** ρ specifies that the value v should be stored in the region ρ. Given the source program above, a compiler can infer that the integer 2 is used only in that expression, so that it inserts **letregion** ρ_2 **in** $\cdots$ **end**. This transformation (which inserts **letregion** ρ $\cdots$ and **at** ρ) is called *region inference* [23].

Region-based memory management has several advantages over conventional memory management schemes. First, it is *safe*, compared with manual memory management using free/malloc in C. Second, it can often deallocate memory cells earlier than conventional, pointer-tracing garbage collection (in the sense that memory cells are deallocated at the end of the letregion construct, while garbage collection is invoked only periodically). Since the original region inference is an extension of the ML type inference, however, it was not clear how to apply the region-based memory management to programming languages other than ML, especially dynamically-typed programming languages such as Scheme [14]. In this paper, we show that the region-based memory management can be applied to dynamically-typed languages by combining region inference and soft typing [5].

We explain the main idea below. First, we review ideas of the original region inference. Under region inference, ordinary types are annotated with region information. For example, the type **int** of integers is replaced by $(\mathbf{int}, \rho)$, which describes integers *stored in region* ρ. Similarly, the function type $\mathbf{int} \rightarrow \mathbf{int}$ is extended to $((\mathbf{int}, \rho_1) \xrightarrow{\varphi} (\mathbf{int}, \rho_2), \rho_3)$, which describes a function stored in region ρ_3 that takes an integer stored in ρ_1 as an argument, accesses regions in φ when it is called, and returns an integer stored in ρ_2. By performing type inference for those extended types, a compiler can statically infer in which region each value is stored and which region is accessed when each expression is evaluated. Using that information, a compiler statically inserts the **letregion** construct. For example, the expression above is given a type $(\alpha \xrightarrow{\{\rho_3\}} (\mathbf{int}, \rho_1), \rho_4)$, where α is an arbitrary type. Using this type, a compiler infers that when the function is applied at execution time, only the region ρ_3 may be accessed and an integer stored in region ρ_1 is returned. Therefore, the compiler can determine that the region ρ_2 is used only in this expression, and insert **letregion** ρ_2 **in** $\cdots$.

As described above, region inference is an extension of ML type inference, so that it cannot be immediately applied to dynamically-typed language. We solve this problem by using the idea of soft typing [5].[1] We construct a new region-

[1] An alternative way would be to translate scheme programs into ML by preparing the following datatype:

```
datatype scm_val = Int of int | Pair of scm_val * scm_val
                             | Fun of scm_val → scm_val | ...
```

It does not work well, since too many values are put into the same region. For example, consider (**if** a **then** $\lambda x.x + 1$ **else** 2). Then, argument and return values of $\lambda x.x + 1$ would be put into the same region as that of 2.

annotated type system which includes union types and recursive types. Using union and recursive types, for example, an expression (**if** a **then** $\lambda x.x$ **else** 1), which may return either a function or an integer, can be given a region-annotated type $(\mathbf{int}, \rho_1) \vee (\tau_1 \xrightarrow{\varphi} \tau_2, \rho_3)$, which means that the expression returns either an integer stored in ρ_1 or a function stored in ρ_3. Using this kind of type, a compiler can translate (**if** a **then** $\lambda x.x$ **else** 1)2 into:

$$\begin{array}{l} \mathbf{letregion}\ \rho_1, \rho_3\ \mathbf{in} \\ \quad (\mathbf{if}\ a\ \mathbf{then}\ (\lambda x.x\ \mathbf{at}\ \rho_3)\ \mathbf{else}\ 1\ \mathbf{at}\ \rho_1)(2\ \mathbf{at}\ \rho_2) \end{array}$$

We have constructed the region-type system hinted above for a core language of Scheme, and proved its soundness. We have also implemented a prototype region inference system for Scheme. In a more general perspective, one of the main contributions of this work is to show that type-based analyses (which have originally been developed for statically-typed languages) can be applied also to dynamically-typed languages by using the idea of soft typing.

The rest of this paper is organized as follows. In Section 2, we introduce a target language of our region inference and define its operational semantics. In Sections 3 and 4, we introduce a region-type system for the target language, and prove its soundness. In Section 5, we sketch a region inference algorithm. In Section 6, we discuss extensions of our target language to deal with full Scheme. In Section 7, we report the result of preliminary experiments on our region inference system. Section 8 discusses related work. Section 9 concludes.

2 Target Language

In this section, we define the syntax and the semantics of the target language of our region inference. It is a λ-calculus extended with constructs for manipulating regions (**letregion** ρ **in** $\cdots$, **at** ρ, etc.). Note that programmers need only to write ordinary functional programs: the constructs for regions are automatically inserted by our region inference described in later sections.

2.1 Syntax

Definition 2.1 [Expressions]: The set of *expressions*, ranged over by e, is given by:

$$\begin{array}{rcl} e\ \text{(expressions)} & ::= & x \mid n\ \mathbf{at}\ \rho \mid \lambda x.e\ \mathbf{at}\ \rho \mid e_1 e_2 \\ & \mid & \mathbf{let}\ f = \mathbf{fix}(f, \Lambda\boldsymbol{\varrho}.(\lambda x.e_1\ \mathbf{at}\ \rho))\ \mathbf{at}\ \rho'\ \mathbf{in}\ e_2 \\ & \mid & f[\boldsymbol{\rho}] \mid \mathbf{if0}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \\ & \mid & \mathbf{letregion}\ \varrho\ \mathbf{in}\ e \\ & \mid & v \mid v[\boldsymbol{\rho}] \\ v\ \text{(run-time values)} & ::= & \langle n \rangle_\rho \mid \langle \lambda x.e \rangle_\rho \mid \langle \mathbf{fix}(f, \Lambda\boldsymbol{\varrho}.(\lambda x.e\ \mathbf{at}\ \rho)) \rangle_{\rho'} \\ \rho\ \text{(regions)} & ::= & \varrho \mid \bullet \end{array}$$

Here, x ranges over a countably infinite set of variables, and n ranges over the set of integers. ϱ ranges over a countably infinite set of region variables. $\boldsymbol{\rho}$ represents a sequence $\rho_1, \ldots, \rho_n$.

The expressions given above includes those for representing run-time values (ranged over by v): they have been borrowed from the formalization of Calcagno et al. [4]. An expression n **at** ρ stores an integer n in region ρ and returns (a pointer to) the integer. A region ρ is either a live region (denoted by ϱ) or a dead region $\bullet$ (that has been already deallocated). Our type system presented in the next section guarantees that n **at** $\bullet$ is never executed. $\lambda x.e$ **at** ρ stores a closure $\lambda x.e$ in region ρ and returns a pointer to it. An expression $e_1 e_2$ applies e_1 to e_2. An expression **let** $f = \mathbf{fix}(f, \Lambda\boldsymbol{\varrho}.(\lambda x.e_1$ **at** $\rho))$ **at** ρ' **in** e_2 stores in region ρ' a recursive, region-polymorphic [23] function f that takes regions and a value as an argument, binds them to $\boldsymbol{\varrho}$ and x, and evaluates e_1; it then binds f to the function and evaluates e_2. An expression $f[\boldsymbol{\rho}]$ applies the region-polymorphic function f to $\boldsymbol{\rho}$. **if0** e_1 **then** e_2 **else** e_3 evaluates e_2 if the value of e_1 is 0, and evaluates e_3 otherwise. **letregion** ρ **in** e creates a new region and binds ρ to the new region; it then evaluates e, deallocates the region ρ, and evaluates to the value of e. Run-time values $\langle n\rangle_\rho$, $\langle \lambda x.e\rangle_\rho$ and $\langle \mathbf{fix}(f, \Lambda\boldsymbol{\varrho}.(\lambda x.e$ **at** $\rho))\rangle_{\rho'}$ denote pointers to an integer, a closure, and a region-polymorphic function respectively. The difference between $\langle n\rangle_\rho$ and n **at** ρ is that the former has already been allocated, so that evaluating it does not cause any memory access, while evaluation of the latter causes an access to the region ρ.

The bound and free variables of e are defined in a customary manner: x is bound in $\lambda x.e$, $f, \boldsymbol{\varrho}$, and x are bound in $\mathbf{fix}(f, \Lambda\boldsymbol{\varrho}.(\lambda x.e_1$ **at** $\rho))$, and ϱ is bound in **letregion** ϱ **in** e. We assume that α-conversion is implicitly performed as necessary, so that all the bound variables are different from each other and from free variables.

2.2 Operational Semantics

We define the operational semantics of our target language, following the formalization of Calcagno et al. [4].

Definition 2.2 [Evaluation Contexts]: The set of *evaluation contexts*, ranged over by E, is given by:

$$\begin{aligned} E ::= & \ [\,] \mid Ee \mid vE \mid \mathbf{if0}\ E\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \\ & \mid \ \mathbf{letregion}\ \varrho\ \mathbf{in}\ E \end{aligned}$$

We write $E[e]$ for the term obtained by replacing $[\,]$ in E with e.

Definition 2.3 [Reduction]: The reduction relation $e \longrightarrow e'$ is the least relation that satisfies the rules in Figure 1.

The relation $e \longrightarrow e'$ means that e is reduced to e' on one step. As in [4], function applications are carried out by using substitutions, so that the identity of each pointer is lost. For example, we cannot tell whether or not two occurrences of $\langle 1\rangle_\rho$ point to the same location. This does not cause a problem in our target language, since there is no primitive for comparing or updating pointers. In the rule R-Reg, region deallocation is modeled by replacement of

$$
\begin{array}{cr}
E[n \textbf{ at } \varrho] \longrightarrow E[\langle n\rangle_\varrho] & \text{(R-INT)}\\
E[\lambda x.e \textbf{ at } \varrho] \longrightarrow E[\langle \lambda x.e\rangle_\varrho] & \text{(R-ABS)}\\
E[\langle \lambda x.e\rangle_\varrho v] \longrightarrow E[[v/x]e] & \text{(R-APP)}\\
\begin{array}{c}E[\langle \mathbf{fix}(f, \Lambda\boldsymbol{\varrho}.(\lambda x.e \textbf{ at } \rho))\rangle_{\varrho'}[\boldsymbol{\rho}]]\\ \longrightarrow E[\langle \lambda x.[\langle \mathbf{fix}(f, \Lambda\boldsymbol{\varrho}.(\lambda x.e \textbf{ at } \rho))\rangle_{\varrho'}/f][\boldsymbol{\rho}/\boldsymbol{\varrho}]e\rangle_{[\boldsymbol{\rho}/\boldsymbol{\varrho}]\rho}]\end{array} & \text{(R-RAPP)}\\
\begin{array}{c}E[\textbf{let } f = \mathbf{fix}(f, \Lambda\boldsymbol{\varrho}.(\lambda x.e_1 \textbf{ at } \rho)) \textbf{ at } \rho' \textbf{ in } e_2]\\ \longrightarrow E[[\langle \mathbf{fix}(f, \Lambda\boldsymbol{\varrho}.(\lambda x.e_1 \textbf{ at } \rho))\rangle_{\rho'}/f]e_2]\end{array} & \text{(R-FIX)}\\
E[\textbf{if0 } \langle 0\rangle_\varrho \textbf{ then } e_1 \textbf{ else } e_2] \longrightarrow E[e_1] & \text{(R-IFT)}\\
E[\textbf{if0 } \langle n\rangle_\varrho \textbf{ then } e_1 \textbf{ else } e_2] \longrightarrow E[e_2] \quad (\text{if } n \neq 0) & \text{(R-IFF)}\\
E[\textbf{letregion } \varrho \textbf{ in } v] \longrightarrow E[[\bullet/\varrho]v] & \text{(R-REG)}
\end{array}
$$

Fig. 1. Reduction rules

a region variable with the dead region $\bullet$. Notice that in each rule, the region accessed in the reduction is denoted by the meta-variable ϱ for live regions, rather than ρ: evaluation gets stuck when the dead region $\bullet$ is accessed.

Example 2.4: Let us consider:

$$\textbf{letregion } \varrho_1, \varrho_5 \textbf{ in } (\lambda x.(\lambda y.(\textbf{letregion } \varrho_3 \textbf{ in } e\ x) \textbf{ at } \varrho_2)) \textbf{ at } \varrho_1)(1 \textbf{ at } \varrho_5)$$

where $e = (\lambda z.(2 \textbf{ at } \varrho_4) \textbf{ at } \varrho_3)$. This is the program obtained by applying region inference to the source program $(\lambda x.(\lambda y.(\lambda z.2)\ x))1$.

The above program is reduced as follows.

$$
\begin{array}{rl}
& \textbf{letregion } \varrho_1, \varrho_5 \textbf{ in } (\lambda x.(\lambda y.(\textbf{letregion } \varrho_3 \textbf{ in } e\ x) \textbf{ at } \varrho_2)) \textbf{ at } \varrho_1)(1 \textbf{ at } \varrho_5)\\
\longrightarrow & \textbf{letregion } \varrho_1, \varrho_5 \textbf{ in } \langle \lambda x.(\lambda y.(\textbf{letregion } \varrho_3 \textbf{ in } e\ x) \textbf{ at } \varrho_2))\rangle_{\varrho_1}(1 \textbf{ at } \varrho_5)\\
\longrightarrow & \textbf{letregion } \varrho_1, \varrho_5 \textbf{ in } \langle \lambda x.(\lambda y.(\textbf{letregion } \varrho_3 \textbf{ in } e\ x) \textbf{ at } \varrho_2))\rangle_{\varrho_1}\langle 1\rangle_{\varrho_5}\\
\longrightarrow & \textbf{letregion } \varrho_1, \varrho_5 \textbf{ in } \lambda y.(\textbf{letregion } \varrho_3 \textbf{ in } e\ \langle 1\rangle_{\varrho_5}) \textbf{ at } \varrho_2)\\
\longrightarrow & \lambda y.(\textbf{letregion } \varrho_3 \textbf{ in } e\ \langle 1\rangle_\bullet) \textbf{ at } \varrho_2)
\end{array}
$$

The result contains a value $\langle 1\rangle_\bullet$ stored in the dead region $\bullet$, but it does not cause a problem since e does not access the value.

3 Type System

In this section, we present a type system for the target language introduced in the previous section. The type system guarantees that every well-typed program never accesses dead regions. So, the problem of region inference is reduced to that of inserting "**letregion** ρ **in** $\cdots$" and " **at** ρ" so that the resulting program is well-typed in the type system (which can be done through type inference).

3.1 Syntax of Types

Definition 3.1 [Types]: The set of *types*, ranged over by τ, is given by:

$$\begin{array}{lll} \mu \text{ (atomic types)} & ::= (\mathbf{num}, \rho) \mid (\tau_1 \xrightarrow{\varphi} \tau_2, \rho) \\ \varphi \text{ (effects)} & ::= \xi \mid \{\rho_1, \dots, \rho_n\} \mid \varphi_1 \cup \varphi_2 \\ \tau \text{ (types)} & ::= r \mid \mathbf{rec}\ r.\mu_1 \vee \cdots \vee \mu_n \\ & \quad \mid \mathbf{rec}\ r.\mu_1 \vee \cdots \vee \mu_n \vee \alpha \\ \pi \text{ (type schemes)} & ::= \forall \boldsymbol{\varrho}^{\varphi}.\forall \boldsymbol{\alpha}.\forall \boldsymbol{\xi}.\tau \end{array}$$

Here, we assume that there are two sets of type variables. One, which is ranged over by α, is the set of type variables bound by universal quantifiers, and the other, which is ranged over by r, is the set of type variables for expressing recursive types.[2] The meta-variable ξ denotes an effect variable.

An atomic type $(\mathbf{num}, \rho)$ describes an integer stored in region ρ. An atomic type $(\tau_1 \xrightarrow{\varphi} \tau_2, \rho)$ describes a function that is stored in ρ and that takes a value of type τ_1 as an argument, accesses regions in φ, and returns a value of type τ_2.

A type $\mathbf{rec}\ r.\mu_1 \vee \cdots \vee \mu_n$ describes a value whose type is one of $[(\mathbf{rec}\ r.\mu_1 \vee \cdots \vee \mu_n)/r]\mu_1, \dots, [(\mathbf{rec}\ r.\mu_1 \vee \cdots \vee \mu_n)/r]\mu_n$. For example, a value of type $\mathbf{rec}\ r.(\mathbf{num}, \rho) \vee (r \xrightarrow{\varphi} r)$ is either an integer or a function that takes a value of type $\mathbf{rec}\ r.(\mathbf{num}, \rho) \vee (r \xrightarrow{\varphi} r)$ and returns a value of the same type. Here, as in the ordinary soft type system [5], we require that the outermost type constructors of $\mu_1, \dots, \mu_n$ are different from each other.[3] For example, $\mathbf{rec}\ r.(\mathbf{num}, \rho) \vee (\mathbf{num}, \rho')$ is invalid. (The restriction must be respected by substitutions; for example, we disallow the substitution $[(\mathbf{num}, \rho)/\alpha]$ to be applied to $\mathbf{rec}\ r.((\mathbf{num}, \rho') \vee \alpha)$.) When r does not appear in $\mu_1, \dots, \mu_n$, we write $\mu_1 \vee \cdots \vee \mu_n$ for $\mathbf{rec}\ r.\mu_1 \vee \cdots \vee \mu_n$. In $\mathbf{rec}\ r.\mu_1 \vee \cdots \vee \mu_n \vee \alpha$, n can be 0, so that $\mathbf{rec}\ r.\alpha$ (which is abbreviated to α) is also a valid type.

Note that union types $\mu_1 \vee \cdots \vee \mu_n$ are *not* annotated with regions. This is because we use a tag-on-pointer representation of data at run-time, where tags to indicate the shape of each data are embedded in pointers. If a tag is stored in the memory cell instead, the union type should be annotated with a region to express where the tag is stored.

A type scheme $\forall \boldsymbol{\varrho}^{\varphi} \forall \boldsymbol{\alpha} \forall \boldsymbol{\xi}.\tau$ describes a region-polymorphic function. The effect φ is the set of regions that may be accessed when regions are passed to the region-polymorphic function.[4] For example, $\mathbf{fix}(f, \Lambda\rho_1\rho_2.(\lambda x.x\ \mathbf{at}\ \rho_2))$ has a type scheme $(\forall \rho_1 \rho_2{}^{\{\rho_2\}}.((\mathbf{num}, \rho_1) \xrightarrow{\emptyset} (\mathbf{num}, \rho_1), \rho_2)$ (assuming that variable x has an integer type).

[2] This distinction between two kinds of variables is necessary to rule out a type expression like $\mathbf{rec}\ r.((\mathbf{num}, \rho) \vee r)$.

[3] Otherwise, type inference would suffer from explosion of case analyses.

[4] Actually, φ is always a singleton set $\{\rho\}$, so that it is possible to make the effect implicit, as in the original region and effect system [23].

3.2 Typing Rules

A type judgment relation is of the form $\Gamma \vdash e : \tau \ \& \ \varphi$. Intuitively, it means that if e is evaluated under an environment that respects the type environment Γ, the evaluation result has type τ and regions in φ may be accessed during the evaluation. Here, a type environment Γ is a mapping from a finite set of variables to the union of the set of types and the set of pairs of the form (π, ρ) (where π is a type scheme and ρ is a region).

Typing rules are given in Figures 2 and 3. Here, the relation $\tau' \prec \forall \boldsymbol{\alpha} \forall \boldsymbol{\xi}.\tau$ used in T-RApp and T-VRApp means that there exist $\boldsymbol{\tau''}$ and $\boldsymbol{\varphi}$ such that $\tau' = [\boldsymbol{\tau''}/\boldsymbol{\alpha}][\boldsymbol{\varphi}/\boldsymbol{\xi}]\tau$. The relation $\mu \subseteq \tau$ means that $\tau = \mathbf{rec}\ r. \cdots \vee \mu' \vee \cdots$ and $\mu = [\tau/r]\mu'$ hold for some r and μ'. $\mathbf{fv}(\Gamma)$ and $\mathbf{fv}(\tau)$ denote the sets of free region, type, and effect variables (i.e., those not bound by $\mathbf{rec}\ r.$ or $\forall \boldsymbol{\varrho}^{\varphi}.\forall \boldsymbol{\alpha}.\forall \boldsymbol{\xi}.$) appearing in Γ and τ respectively.

Note that in the rule T-App, e_1 need not be a function, since τ_1 may be $(\mathbf{num}, \rho') \vee (\tau_2 \xrightarrow{\varphi_0} \tau_3, \rho)$. When $e_1 e_2$ is evaluated, e_1 and e_2 are first evaluated and the regions in $\varphi_1 \cup \varphi_2$ may be accessed. After that, if the value of e_1 is a function, then the function is called and the regions in $\varphi_0 \cup \{\rho\}$ may be accessed. Otherwise, the evaluation gets stuck, so that no more region is accessed. So, the effect $\varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\rho\}$ soundly estimates the set of regions that are accessed when $e_1 e_2$ is evaluated, irrespectively of whether the value of e_1 is a function or not. Recall that we use a tag-on-pointer representation of data at run-time, so, no region is accessed when it is checked whether the value of e_1 is a function or not.

Example 3.2: The type judgment:

$$\emptyset \vdash \mathbf{letregion}\ \rho_0, \rho_1, \rho_3\ \mathbf{in}$$
$$(\mathbf{if0}\ n\ \mathbf{at}\ \rho_0\ \mathbf{then}\ (\lambda x.x\ \mathbf{at}\ \rho_3)\ \mathbf{else}\ 1\ \mathbf{at}\ \rho_1)(2\ \mathbf{at}\ \rho_2) : (\mathbf{num}, \rho_2)\ \&\ \{\rho_2\}$$

is derived as follows (here, n is some integer).

First, we can obtain $\emptyset \vdash n\ \mathbf{at}\ \rho_0 : (\mathbf{num}, \rho_0) \& \{\rho_0\}$ and $x : (\mathbf{num}, \rho_2) \vdash x : (\mathbf{num}, \rho_2)\ \&\ \emptyset$ by using the rule T-Int and T-Var. By applying rule T-Abs to the latter, we obtain

$$\emptyset \vdash \lambda x.x\ \mathbf{at}\ \rho_3 : ((\mathbf{num}, \rho_2) \xrightarrow{\emptyset} (\mathbf{num}, \rho_2), \rho_3) \vee (\mathbf{num}, \rho_1) \& \{\rho_3\}.$$

We can also obtain

$$\emptyset \vdash 1\ \mathbf{at}\ \rho_1 : ((\mathbf{num}, \rho_2) \xrightarrow{\emptyset} (\mathbf{num}, \rho_2), \rho_3) \vee (\mathbf{num}, \rho_1) \& \{\rho_1\}$$

by using T-Int. By applying T-If and T-App, we obtain

$$\emptyset \vdash (\mathbf{if0}\ n\ \mathbf{at}\ \rho_0\ \mathbf{then}\ (\lambda x.x\ \mathbf{at}\ \rho_3)\ \mathbf{else}\ 1\ \mathbf{at}\ \rho_1)(2\ \mathbf{at}\ \rho_2)$$
$$: (\mathbf{num}, \rho_2) \& \{\rho_0, \rho_1, \rho_2, \rho_3\}$$

Finally, by using T-Reg, we obtain:

$$\emptyset \vdash \mathbf{letregion}\ \rho_0, \rho_1, \rho_3\ \mathbf{in}$$
$$(\mathbf{if0}\ n\ \mathbf{at}\ \rho_0\ \mathbf{then}\ (\lambda x.x\ \mathbf{at}\ \rho_3)\ \mathbf{else}\ 1\ \mathbf{at}\ \rho_1)(2\ \mathbf{at}\ \rho_2) : (\mathbf{num}, \rho_2)\ \&\ \{\rho_2\}.$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \;\&\; \emptyset}\ (\text{T-Var})$$

$$\frac{(\mathbf{num}, \rho) \subseteq \tau}{\Gamma \vdash n \ \mathbf{at}\ \rho : \tau \;\&\; \{\rho\}}\ (\text{T-Int})$$

$$\frac{\Gamma + \{x \mapsto \tau_1\} \vdash e : \tau_2 \;\&\; \varphi' \qquad \varphi' \subseteq \varphi \qquad (\tau_1 \xrightarrow{\varphi} \tau_2, \rho) \subseteq \tau_3}{\Gamma \vdash \lambda x.e \ \mathbf{at}\ \rho : \tau_3 \;\&\; \{\rho\}}\ (\text{T-Abs})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \;\&\; \varphi_1 \qquad (\tau_2 \xrightarrow{\varphi_0} \tau_3, \rho) \subseteq \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \;\&\; \varphi_2}{\Gamma \vdash e_1 e_2 : \tau_3 \;\&\; \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\rho\}}\ (\text{T-App})$$

$$\frac{\Gamma(f) = (\pi, \rho_f) \qquad \pi = \forall \boldsymbol{\varrho}^{\varphi} \forall \boldsymbol{\alpha} \forall \boldsymbol{\xi}.\tau \qquad \tau' \prec \forall \boldsymbol{\alpha} \forall \boldsymbol{\xi}.[\boldsymbol{\rho}'/\boldsymbol{\varrho}]\tau}{\Gamma \vdash f[\boldsymbol{\rho}'] : \tau' \;\&\; \{\rho_f\} \cup [\boldsymbol{\rho}'/\boldsymbol{\varrho}]\varphi}\ (\text{T-RApp})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \;\&\; \varphi_1 \qquad (\mathbf{num}, \rho) \subseteq \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \;\&\; \varphi_2 \qquad \Gamma \vdash e_3 : \tau_2 \;\&\; \varphi_3}{\Gamma \vdash \mathbf{if0}\ e_1 \ \mathbf{then}\ e_2 \ \mathbf{else}\ e_3 : \tau_2 \;\&\; \varphi_1 \cup \varphi_2 \cup \varphi_3 \cup \{\rho\}}\ (\text{T-If})$$

$$\frac{\Gamma \vdash e : \tau \;\&\; \varphi \qquad \varrho \notin \mathbf{fv}(\Gamma) \cup \mathbf{fv}(\tau)}{\Gamma \vdash \mathbf{letregion}\ \varrho \ \mathbf{in}\ e : \tau \;\&\; \varphi \setminus \{\varrho\}}\ (\text{T-Reg})$$

$$\frac{\begin{array}{c}\pi = \forall \boldsymbol{\varrho}^{\varphi_1} \forall \boldsymbol{\xi}.\tau_1 \qquad \{\boldsymbol{\varrho}, \boldsymbol{\xi}, \boldsymbol{\alpha}\} \cap (\mathbf{fv}(\Gamma) \cup \{\rho_f\}) = \emptyset \\ \Gamma + \{f \mapsto (\pi, \rho_f)\} \vdash \lambda x.e_1 \ \mathbf{at}\ \rho_t : \tau_1 \;\&\; \varphi_1 \\ \pi' = \forall \boldsymbol{\varrho}^{\varphi_1} \forall \boldsymbol{\alpha} \forall \boldsymbol{\xi}.\tau_1 \qquad \Gamma + \{f \mapsto (\pi', \rho_f)\} \vdash e_2 : \tau_2 \;\&\; \varphi_2\end{array}}{\Gamma \vdash \mathbf{let}\ f = \mathbf{fix}(f, \Lambda \boldsymbol{\varrho}.(\lambda x.e_1 \ \mathbf{at}\ \rho_t)) \ \mathbf{at}\ \rho_f \ \mathbf{in}\ e_2 : \tau_2 \;\&\; \{\rho_f\} \cup \varphi_2}\ (\text{T-Fix})$$

Fig. 2. Typing rules for static expressions

$$\frac{\Gamma \vdash v : (\forall \boldsymbol{\varrho}^{\varphi} \forall \boldsymbol{\alpha} \forall \boldsymbol{\xi}.\tau, \rho_f) \qquad \tau' \prec \forall \boldsymbol{\alpha} \forall \boldsymbol{\xi}.[\boldsymbol{\rho}'/\boldsymbol{\varrho}]\tau}{\Gamma \vdash v[\boldsymbol{\rho}'] : \tau' \;\&\; \{\rho_f\} \cup [\boldsymbol{\rho}'/\boldsymbol{\varrho}]\varphi}\ (\text{T-VRapp})$$

$$\frac{(\mathbf{num}, \rho) \subseteq \tau}{\Gamma \vdash \langle n \rangle_{\rho} : \tau \;\&\; \emptyset}\ (\text{T-VInt})$$

$$\frac{\Gamma + \{x \mapsto \tau_1\} \vdash e : \tau_2 \;\&\; \varphi' \qquad \varphi' \subseteq \varphi \qquad (\tau_1 \xrightarrow{\varphi} \tau_2, \rho) \subseteq \tau}{\Gamma \vdash \langle \lambda x.e \rangle_{\rho} : \tau \;\&\; \emptyset}\ (\text{T-VAbs})$$

$$\frac{\begin{array}{c}\pi = \forall \boldsymbol{\varrho}^{\varphi} \forall \boldsymbol{\xi}.\tau \\ \{\boldsymbol{\varrho}, \boldsymbol{\xi}, \boldsymbol{\alpha}\} \cap (\mathbf{fv}(\Gamma) \cup \{\rho_f\}) = \emptyset \\ \Gamma + \{f \mapsto (\pi, \rho_f)\} \vdash \lambda x.e \ \mathbf{at}\ \rho_t : \tau \;\&\; \varphi \\ \pi' = \forall \boldsymbol{\varrho} \forall^{\varphi} \boldsymbol{\alpha} \forall \boldsymbol{\xi}.\tau\end{array}}{\Gamma \vdash \langle \mathbf{fix}(f, \Lambda \boldsymbol{\varrho}.(\lambda x.e \ \mathbf{at}\ \rho_t)) \rangle_{\rho_f} : (\pi', \rho_f) \;\&\; \emptyset}\ (\text{T-VFix})$$

Fig. 3. Typing rules for dynamic expressions

Example 3.3: An expression corresponding to a source program **if0** 1 **then** $\lambda x.x$ **else** $\lambda x.2$ is typed as follows.

$$\begin{array}{l}\emptyset \vdash \mathbf{letregion}\ \rho_0 \ \mathbf{in} \\ \qquad \mathbf{if0}\ 1 \ \mathbf{at}\ \rho_0 \ \mathbf{then}\ \lambda x.x \ \mathbf{at}\ \rho_1 \ \mathbf{else}\ \lambda x.(2 \ \mathbf{at}\ \rho_2) \ \mathbf{at}\ \rho_1 : \\ \qquad\qquad (((\mathbf{num}, \rho_2) \vee \alpha) \xrightarrow{\{\rho_1, \rho_2\}} ((\mathbf{num}, \rho_2) \vee \alpha), \rho_1) \;\&\; \{\rho_1\}\end{array}$$

The then-part must have a type of the form $(\alpha \xrightarrow{\varphi_1} \alpha, \rho_1)$ where $\{\rho_1\} \subseteq \varphi_1$ and the else-part must have a type of the form $(\beta \xrightarrow{\varphi_2} (\mathbf{num}, \rho_2), \rho_1)$ where $\{\rho_1, \rho_2\} \subseteq \varphi_2$. The type of the whole if-expression above can be obtained by unifying those types.

4 Properties of the Type System

The soundness of the type system is guaranteed by Theorems 4.1 and 4.2 given below. Theorem 4.1 implies that a well-typed, closed (i.e., not containing free variables) expression does not access a deallocated region immediately. Theorem 4.2 implies that the well-typedness of an expression is preserved by reduction. These theorems together imply that a well-typed, closed expression never accesses a deallocated region. Our proof is based on the syntactic type soundness proof of Calcagno et al. [4], and extends it to handle union/recursive types and polymorphism.

Theorem 4.1: Suppose $\emptyset \vdash e : \tau \ \& \ \varphi$, and e is one of the following forms:

- $E[n \textbf{ at } \rho]$
- $E[\lambda x.e \textbf{ at } \rho]$
- $E[\langle \lambda x.e \rangle_\rho v]$
- $E[\langle \mathbf{fix}(f, \Lambda \boldsymbol{\varrho}.(\lambda x.e \textbf{ at } \rho'))\rangle_\rho [\boldsymbol{\rho''}]]$
- $E[\textbf{let } f = \mathbf{fix}(f, \Lambda \boldsymbol{\varrho}.(\lambda x.e_1 \textbf{ at } \rho')) \textbf{ at } \rho \textbf{ in } e_2]$
- $E[\textbf{if0 } \langle n \rangle_\rho \textbf{ then } e_1 \textbf{ else } e_2]$

If $\bullet \notin \varphi$, then $\rho \neq \bullet$. In the fourth case, $[\boldsymbol{\rho''}/\boldsymbol{\varrho}]\rho' \neq \bullet$ also holds.

Theorem 4.2 [Subject Reduction]: If $\Gamma \vdash e : \tau \ \& \ \varphi$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau \ \& \ \varphi'$ for some φ' such that $\varphi' \subseteq \varphi$.

Proofs of the theorems above are found in the full version of this paper [19].

Note that the type system does not guarantee that evaluation of a well-typed program never gets stuck: since the target of our study is a dynamically-typed language like Scheme, our type system does allow an expression like **if0** $\langle \lambda x.e \rangle_\rho$ **then** e_1 **else** e_2. In fact, our type system can type *any* source program, as stated in Theorem 4.5 below.

Definition 4.3 [Source Programs]: The set of *source programs*, ranged over by M, is given by:

$$\begin{aligned} M ::= {} & x \mid f \mid n \mid \lambda x.M \mid M_1 M_2 \\ & \mid \textbf{let } f = \mathbf{fix}(f, \lambda x.M_1) \textbf{ in } M_2 \mid \textbf{if0 } M_1 \textbf{ then } M_2 \textbf{ else } M_3 \end{aligned}$$

Definition 4.4 [Region Erasure]: The *region erasure function* $(\cdot)^\sharp$ is a partial mapping from the set of expressions to the set of source programs, defined by:

$$
\begin{aligned}
&x^\sharp = x\\
&(n \textbf{ at } \rho)^\sharp = n\\
&(\lambda x.e \textbf{ at } \rho)^\sharp = \lambda x.e^\sharp\\
&(\textbf{let } f = \textbf{fix}(f, \Lambda\boldsymbol{\varrho}.(\lambda x.e_1 \textbf{ at } \rho)) \textbf{ at } \rho' \textbf{ in } e_2)^\sharp = \textbf{let } f = \textbf{fix}(f, \lambda x.e_1^\sharp) \textbf{ in } e_2^\sharp\\
&(f[\boldsymbol{\rho}])^\sharp = f\\
&(\textbf{if0 } e_1 \textbf{ then } e_2 \textbf{ else } e_3)^\sharp = \textbf{if0 } e_1^\sharp \textbf{ then } e_2^\sharp \textbf{ else } e_3^\sharp\\
&(\textbf{letregion } \varrho \textbf{ in } e)^\sharp = e^\sharp
\end{aligned}
$$

Theorem 4.5: For any closed source program M, there exist e, τ and φ such that $\emptyset \vdash e : \tau \ \& \ \varphi$ and $e^\sharp = M$.

Proof. Let τ be $\textbf{rec } r.((\textbf{num}, \rho_G) \vee (r \xrightarrow{\{\rho_G\}} r, \rho_G))$. Let us define a function $(\cdot)^\flat$ from the set of source programs to expressions by:

$$
\begin{aligned}
&x^\flat = x\\
&f^\flat = f[\epsilon]\\
&n^\flat = n \textbf{ at } \rho_G\\
&(\lambda x.M)^\flat = \lambda x.e^\flat \textbf{ at } \rho_G\\
&(\textbf{let } f = \textbf{fix}(f, \lambda x.M_1) \textbf{ in } M_2)^\flat =\\
&\qquad\qquad \textbf{let } f = \textbf{fix}(f, \Lambda\epsilon.(\lambda x.M_1^\flat \textbf{ at } \rho_G)) \textbf{ at } \rho_G \textbf{ in } M_2^\flat\\
&(\textbf{if0 } M_1 \textbf{ then } M_2 \textbf{ else } M_3)^\flat = \textbf{if0 } M_1^\flat \textbf{ then } M_2^\flat \textbf{ else } M_3^\flat
\end{aligned}
$$

Here, ϵ denotes the empty sequence of regions. The idea of the above translation is to use ρ_G as a special region that is never deallocated and where all values are stored. It is easy to check that $\emptyset \vdash M^\flat : \tau \ \& \ \varphi$ holds for either $\varphi = \{\rho_G\}$ or $\varphi = \emptyset$. (In the derivation, assign the type τ to every variable ranged over by x, and assign the polymorphic type $(\forall\epsilon^{\{\rho_G\}}.\tau, \rho_G)$ to every variable ranged over by f.)

The above theorem guarantees that for any source program, there is at least one valid (i.e., well-typed) region-annotated expression. Of course, the expression constructed in the proof above is not a good annotation, since no region is deallocated. How to find a *good* annotation is discussed in the next section.

5 Region Inference

In this section, we show how to perform region inference, i.e., transform a source program (without constructs for regions) into a program of the target language defined in section 2. The region inference is carried out in the following steps.

1. Based on the typing rules defined in Section 3, a standard type (types without regions and effects) is inferred for each expression. This can be carried out by using the soft type inference algorithm [5].
2. Fresh region variables and effect variables are added to the types inferred above.

3. Based on the typing rules in Section 3, the actual values of region variables and effect variables are computed. During this, some region and effect variables introduced in the previous step are unified. This can be carried out in a way similar to the ordinary region inference [22]. Finally, **letregion** is inserted in the place where the side condition of T-Reg is met. (Actually, inference of regions and effects and insertion of **letregion** have to be carried out in an interleaving manner to handle region polymorphism [22].)

Note that the third step is almost the same as the original region inference algorithm. Although our typing rules are a little more complex because of union types and recursive types, that difference is absorbed in the first step, where the *shape* of union types and recursive types are determined. For example, after the first phase, the type τ_3 in the rule T-Abs is instantiated to a type of the form $\mathbf{rec}\ r.((\tau_1' \xrightarrow{\varphi'} \tau_2', \rho') \vee \cdots)$, so that it is sufficient to solve the unification constraint $(\tau_1 \xrightarrow{\varphi} \tau_2, \rho) = (\tau_1' \xrightarrow{\varphi'} \tau_2', \rho')$ in the third step, as in the original region inference algorithm.

Since the actual algorithm (especially, the third step: see [22]) is rather complex, we sketch it here only through examples. Please consult the full version for more details [19].

Example 5.1: Consider the expression:

$$(\mathbf{if0}\ n\ \mathbf{then}\ (\lambda x.x)\ \mathbf{else}\ 1)2.$$

Here, n is an integer. Region inference for this expression is performed as follows.

First, the standard type (without regions) of the expression is inferred as $\mathbf{num} \vee (\mathbf{num} \longrightarrow \mathbf{num})$. Then, region and effect variables are inserted, as $(\mathbf{num}, \rho_1) \vee ((\mathbf{num}, \rho_2) \xrightarrow{\emptyset} (\mathbf{num}, \rho_2), \rho_3)$. Using this type, the effect of the whole expression is inferred as $\{\rho_0, \rho_1, \rho_2, \rho_3\}$. The regions ρ_0, ρ_1 and ρ_3 do not appear in the type environment (which is empty) and the type of the returned value $(\mathbf{num}, \rho_2)$, so that **letregion** can be inserted as follows.

$$\begin{array}{l}\mathbf{letregion}\ \rho_0, \rho_1, \rho_3\ \mathbf{in}\\ \quad (\mathbf{if0}\ n\ \mathbf{at}\ \rho_0\ \mathbf{then}\ (\lambda x.x\ \mathbf{at}\ \rho_3)\ \mathbf{else}\ 1\ \mathbf{at}\ \rho_1)(2\ \mathbf{at}\ \rho_2)\end{array}$$

Example 5.2: Let us consider a recursive function:

$$\mathbf{fix}(f, \lambda x.\mathbf{if0}\ x\ \mathbf{then}\ x\ \mathbf{else}\ f(x-1)-1).$$

(Here, we have extended the language with the operation '−.') In the first phase, the type $\mathbf{num} \rightarrow \mathbf{num}$ is inferred. In the second phase the function is tentatively given a type[5]

[5] As in [22], we do not consider quantifications over secondary region and effect variables to ensure termination of the algorithm.

$$\forall \rho_1, \rho_2, \rho_3^{\{\rho_3\}}.\forall\xi.((\mathbf{num}, \rho_1) \xrightarrow{\xi} (\mathbf{num}, \rho_2), \rho_3)$$

and the program is annotated as follows.

$$\begin{array}{l}\mathbf{fix}(f, \Lambda\rho_1, \rho_2, \rho_3.(\lambda x.\mathbf{if0}\ x\ \mathbf{then}\ x \\ \quad \mathbf{else}\ (f[\rho_4, \rho_5, \rho_6](x - (1\ \mathbf{at}\ \rho_7)\ \mathbf{at}\ \rho_8) - (1\ \mathbf{at}\ \rho_9))\ \mathbf{at}\ \rho_{10})\ \mathbf{at}\ \rho_3)\end{array}$$

In the third phase, assuming the tentative type above for f, we perform region inference for the function body, unify some region variables and insert **letregion**. For example, from the type of the then-part and the else-part, it must be the case that $\rho_1 = \rho_2 = \rho_{10}$. From the call of f, we also have $\rho_4 = \rho_8$. From this, we obtain the following refined expression:

$$\begin{array}{l}\mathbf{fix}(f, \Lambda\rho_1, \rho_3.(\lambda x.(\mathbf{if0}\ x\ \mathbf{then}\ x\ \mathbf{else}\ \mathbf{letregion}\ \rho_4, \rho_6, \rho_9\ \mathbf{in} \\ \quad (f[\rho_4, \rho_6](\mathbf{letregion}\ \rho_7\ \mathbf{in}\ (x - (1\ \mathbf{at}\ \rho_7))\ \mathbf{at}\ \rho_4) - (1\ \mathbf{at}\ \rho_9))\ \mathbf{at}\ \rho_1)\ \mathbf{at}\ \rho_3))\end{array}$$

and its type: $\forall \rho_1, \rho_3^{\{\rho_3\}}.\forall\xi.((\mathbf{num}, \rho_1) \xrightarrow{\xi\cup\{\rho_1,\rho_3\}} (\mathbf{num}, \rho_1), \rho_3)$. We repeat this refinement step until the result converges. In the case above, the above program is a final one.

6 Language Extensions

In this section, we show how to extend the target language defined in Section 2 to support full Scheme.

Cons Cells. We introduce cons cells by adding a new atomic type $(\tau_1 \times \tau_2, \rho)$, which describes a cons cell that is stored in ρ and consists of a car-element of type τ_1 and a cdr-element of type τ_2. We can deal with **set-car!** and **set-cdr!** by assigning the following types to them:

$$\begin{array}{l}\mathbf{set\text{-}car!} : \forall \rho_1\rho_2\rho_3^{\{\rho_3\}}.\forall\alpha_1\alpha_2\alpha_3.\forall\xi_1\xi_2. \\ \qquad ((\alpha_1 \times \alpha_2, \rho_1) \xrightarrow{\{\rho_2\}\cup\xi_1} (\alpha_1 \xrightarrow{\{\rho_1\}\cup\xi_2} \alpha_3, \rho_2), \rho_3) \\ \mathbf{set\text{-}cdr!} : \forall \rho_1\rho_2\rho_3^{\{\rho_3\}}.\forall\alpha_1\alpha_2\alpha_3.\forall\xi_1\xi_2. \\ \qquad ((\alpha_1 \times \alpha_2, \rho_1) \xrightarrow{\{\rho_2\}\cup\xi_1} (\alpha_2 \xrightarrow{\{\rho_1\}\cup\xi_2} \alpha_3, \rho_2), \rho_3)\end{array}$$

To ensure the type soundness, polymorphic types are not assigned to cons cells. For example, $\forall\alpha.((\mathbf{num}, \rho) \times (\alpha \xrightarrow{\varphi} \alpha, \rho'), \rho'')$ is not allowed. Vector types and other complex data types can be introduced in the same way.

set! We translate **set!** into ML-like operations on reference cells and then perform region inference in the same way as that for ML [23]. To perform the translation, we first perform a program analysis to find all the variables whose values might be updated by **set!**, and then replace all the accesses to those variables with ML-like operations on reference cells. For example, (**let** ((x (+ a 1))) ... (**set!** x 2)) is translated to (**let** ((x (**ref** (+ a 1)))) ... (:= x 2)). Here, **ref** v is a primitive for creating a reference cell storing v and returns the pointer to it, and $v_1 := v_2$ is a primitive that stores v_2 in the reference cell v_1.

call/cc It seems difficult to deal with call-with-current-continuation (**call/cc**) in a completely static manner. (In fact, the region inference system for ML does not handle **call/cc**, either.) One (naive) way to deal with **call/cc** might be, when **call/cc** is invoked at run-time, to move the contents of the stack and the heap space reachable from the stack to a global region, so that they can be only collected by standard garbage collection, not by region-based memory management. An alternative way would be to first perform CPS-transformation, and then perform the region inference.

7 Implementation

Based on the type system introduced in Section 3, we have implemented a region inference system for Scheme. Cons cells and **set!** discussed in Section 6 have been already supported, but call-with-current-continuation has not been supported yet. The system transforms a source program written in Scheme into a region-annotated program, whose core syntax has been given in Section 2, and then translates it into C language. For the experiments reported below, we have inserted instructions for monitoring memory usage in the region operation library. Our implementation is available at

`http://www.yl.is.s.u-tokyo.ac.jp/~ganat/research/region/`

We have tested our region inference system for several programs, and confirmed that the translated programs run correctly. For example, the following program (which computes the number of leaves of a binary tree):

```
(define (leafcount t)
  (if (pair? t) (+ (leafcount (car t)) (leafcount (cdr t))) 1))
```

has been automatically translated by our system into

```
(define leafcount
   (reglambda (r60 r57 r59 r58)
     (lambda (v2)
         (if (letregion (r62) (pair?[r57 r62] v2))
             (letregion (r67 r69 r88)
                 (+[r88 r67 r59 r69]
                     (letregion (r73)
                        (leafcount[r73 r57 r88 r76]
                        (letregion (r82) (car[r57 r82] v2))))
                     (letregion (r86)
                        (leafcount[r86 r57 r88 r89]
                        (letregion (r95 ) (cdr[r57 r95] v2))))))
              1 at r59))
       at r60)
at r52)
```

Here, `reglambda` creates a region-polymorphic function. The instruction `leafcount[r73 r57 r88 r76]` applies the region-polymorphic function `leafcount` to region parameters `r73`, `r57`, `r88`, and `r76`. The instruction `1 at r1` puts the number `1` into region `r1`. Note that during the translation, the tree argument `t` above is given a type of the form $\mathbf{rec}\ r.(r \times r, \rho_1) \vee \alpha$, which contains recursive and union types, so that it cannot be handled by the original region inference [23] for ML unless a programmer defines a tree type using a datatype declaration.

The result of experiments is summarized in Table 1. The table shows execution time, the maximum heap size, and the total size of allocated memory cells. To evaluate the effectiveness of the region-based memory management, we have also measured the execution time and the heap size of a system with garbage collection, by turning off the region inference phase of our complier and running the compiled code with Boehm GC library 6.2. The executioin time and the heap size of our region-based system are listed in the rows marked "Region," while those of GC are listed in the rows "GC."

Table 1. Results of the Experiments

	Program Size	Time (msec.)		Heap Size (KBytes)		Total Memory
program	(Lines)	Region	GC	Region	GC	Allocation (KBytes)
`Fib`	9	51.6	27.0	4.5	49.2	323.1
`Ackermann`	7	59.9	32.4	18.0	89.1	399.7
`Tree`	16	22.2	13.6	6.2	66.6	177.6
`Array`	16	347.8	196.9	280.8	287.7	2342.4
`QuickSort`	100	793.5	526.4	695.5	693.2	5272.1
`Tak`	23	428.0	255.2	465.1	66.6	3733.8
`Div`	54	579.3	399.6	32.2	1219.6	4085.4
`Deriv`	65	1018.1	680.2	3327.5	5146.6	7245.4
`Destruct`	72	7967.9	4432.0	10960.9	1219,6	59259.5
`RayTracing`	1627	2371.5	1522.8	157.8	287.7	14155.7

Programs `Array`, `Tak`, `Div`, `Deriv`, `Destruct` have been taken from Gabriel Scheme benchmarks [8]. `Tree` is the program given above to count leafs, with a tree of size 18 given as an input. `RayTracing` is a program for ray tracing. `QuickSort` is a program for quick sort. `Fib` and `Ackermann` calculate Fibonacci and Ackermann number, respectively. The difference between the maximum heap size and the total size of allocated memory shows the effectiveness of our region inference. For example, for `RayTracing`, the total size of allocated memory was 14.2 MBytes, but the required heap space was 2.4 MBytes.

As for the comparison with the GC library, for some programs, the space efficiency of our region-based memory management is significantly better than that of the GC library. For the program `Tak`, however, garbage collection works better. These results suggest that combination of the two memory management schemes may be attractive [12]. As for the time efficiency, our current region-based memory management cannot compete with the GC library. Optimizations for the region-based memory management such as storage mode analysis [1, 2] would be necessary to make the region-based memory management competitive with garbage collection.

8 Related Work

Region-based memory management has been applied to programming languages other than ML [3, 6, 7, 9–11, 17, 18] but most of them rely on programmers' annotations on region instructions (such as "**letregion**" and "**at** ρ"). Only a few of them, which are discussed below, support region inference (i.e., automatic insertion of region instructions). Makholm [17, 18] studied region inference for Prolog. As in our work, his region inference algorithm is based on soft typing, but technical details seem to be quite different since Prolog does not have higher-order functions (hence no need for effects) and instead has logical variables. Deters and Cytron [7] have proposed an algorithm to insert memory allocation/deallocation instructions (similar to region instructions) for Real-Time Java. Their method is based on run-time profiling, so that there seems to be no guarantee that the instructions are inserted correctly. Grossman et al. [11] has proposed a type system for region-based memory management for Cyclone (a type-safe dialect of C). In Cyclone, programmers have to explicitly insert code for manipulating regions, but some of the region annotations are inferred using some heuristics.

The idea of applying type-based program analyses to dynamically-typed programming languages by using soft typing might be a kind of folklore. In fact, Rehof has hinted on that in 1995 [21]. To the authors' knowledge, however, our work is the first to give a concrete formalization of a type-based program analysis for dynamically-typed functional languages and prove the soundness.

9 Conclusion

We have proposed a new region-type system for a dynamically-typed language, and proved its correctness. Based on the type system, we have also implemented a prototype region inference system for Scheme and tested it for several Scheme programs.

Support for call-with-current-continuation is left for future work. To make the region-based memory management more effective, we also need to incorporate several analyses such as region size inference [2]. Combination with other type-based methods for memory management [16] would also be interesting.

The general approach of this work – using soft types to apply a type-based analysis that has been originally developed for statically-typed languages to dynamically-typed languages – seems to be applicable to other type-based analyses such as linear type systems [15, 24], exception analysis [20], and resource usage analysis [13].

References

1. A. Aiken, M. Fahndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proc. of PLDI*, pages 174–185, 1995.
2. L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proc. of POPL*, pages 171–183. ACM Press, January 1996.
3. C. Boyapati, A. Salcianu, W. Beebee, and J. Rinard. Ownership types for safe region-based memory management in Real-Time Java, 2003.
4. C. Calcagno, S. Helsen, and P. Thiemann. Syntactic type soundness results for the region calculus. *Info. Comput.*, 173(2):199–221, 2002.
5. R. Cartwright and M. Fagan. Soft typing. In *Proc. of PLDI*, pages 278–292, 1991.
6. K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proc. of POPL*, pages 262–275, New York, NY, 1999.
7. M. Deters and R. K. Cytron. Automated discovery of scoped memory regions for real-time java. In *Proceedings of ISMM'02*, pages 25–35. ACM Press, 2002.
8. R. Gabriel. Scheme version of the gabriel lisp benchmarks, 1988.
9. D. Gay and A. Aiken. Memory management with explicit regions. In *Proc. of PLDI*, pages 313–323, 1998.
10. D. Gay and A. Aiken. Language support for regions. In *Proc. of PLDI*, pages 70–80, 2001.
11. D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, 2002.
12. N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *Proc. of PLDI*, pages 141–152. ACM Press, 2002.
13. A. Igarashi and N. Kobayashi. Resource usage analysis. To appear in *ACM Trans. Prog. Lang. Syst.* A summary appeared in Proc. of POPL, pages 331–342, 2002.
14. R. Kelsey, W. Clinger, and J. R. (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
15. N. Kobayashi. Quasi-linear types. In *Proc. of POPL*, pages 29–42, 1999.
16. O. Lee, H. Yang, and K. Yi. Inserting safe memory reuse commands into ml-like programs. In *Proceedings of SAS 2003*, volume 2694 of *LNCS*, pages 171–188, 2003.
17. H. Makholm. Region-based memory management in Prolog. Master's thesis, DIKU, University of Copenhagen, 2000.
18. H. Makholm. A region-based memory manager for Prolog. In B. Demoen, editor, *First Workshop on Memory Management in Logic Programming Implementations*, volume CW 294, pages 28–40, CL2000, London, England, 24 2000. Katholieke Universiteit Leuven.
19. A. Nagata, N. Kobayashi, and A. Yonezawa. Region-based memory management for a dynamically-typed language, 2004. Full version, available from `http://www.yl.is.s.u-tokyo.ac.jp/~{}ganat/research/region/`.
20. F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proc. of POPL*, pages 276–290, 1999.
21. J. Rehof. Polymorphic dynamic typing. aspects of proof theory and inferencej. Master's thesis, DIKU, University of Copenhagen, August 1995.
22. M. Tofte and L. Birkedal. A region inference algorithm. *ACM Trans. Prog. Lang. Syst.*, 20(4):724–767, July 1998.

23. M. Tofte and J.-P. Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proc. of POPL*, pages 188–201. ACM Press, January 1994.
24. D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proc. of Functional Programming Languages and Computer Architecture*, pages 1–11, San Diego, California, 1995.