Region-Based Memory Management for a Dynamically-Typed Language

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

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

Abstract. Region-based memory management scheme has been proposed for programming language ML. In this scheme, a compiler statically estimates the live range 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. [19] 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 life time 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

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

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 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* [19].

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. 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 [13]. 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. In the 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, the 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]. We construct a new region-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:

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

We have constructed a 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 Section 3, 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:

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

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. $\tilde{\rho}$ 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** $f = \mathbf{fix}(f, \Lambda\tilde{\varrho}.(\lambda x.e_1 \ \mathbf{at}\ \rho))$ **at** ρ' **in** e_2 stores in region ρ' a recursive, region-polymorphic [19] function f that takes regions and a value as an argument, binds them to $\tilde{\varrho}$ and x, and evaluates e; It then binds f to the function and evaluates e_2. An expression $f[\tilde{\rho}]$ applies the region-polymorphic function f to $\tilde{\rho}$. **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 returns the value of e. Run-time values $\langle n\rangle_\rho$, $\langle \lambda x.e\rangle_\rho$ and $\langle \mathbf{fix}(f, \Lambda\tilde{\varrho}.(\lambda x.e\ \mathbf{at}\ \rho))\rangle_{\rho'}$ 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 region ρ.)

The bound and free variables of e are defined in a customary manner: x is bound in $\lambda x.e$, $f, \tilde{\varrho}$, and x are bound in $\mathbf{fix}(f, \Lambda\tilde{\varrho}.(\lambda x.e_1\ \mathbf{at}\ \rho))$, 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 an 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:

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

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 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.

$$
\begin{array}{cr}
E[n \textbf{ at } \varrho] \longrightarrow E[\langle n\rangle_\varrho] & (\text{R-I}\textsc{nt}) \\
E[\lambda x.e \textbf{ at } \varrho] \longrightarrow E[\langle \lambda x.e\rangle_\varrho] & (\text{R-A}\textsc{bs}) \\
E[\langle \lambda x.e\rangle_\varrho v] \longrightarrow E[[v/x]e] & (\text{R-A}\textsc{pp}) \\
\begin{array}{c} E[\langle \textbf{fix}(f, \Lambda\tilde{\varrho}.(\lambda x.e \textbf{ at } \rho))\rangle_{\varrho'}[\tilde{\rho}]] \\ \longrightarrow E[\langle \lambda x.[\langle \textbf{fix}(f, \Lambda\tilde{\varrho}.(\lambda x.e \textbf{ at } \rho))\rangle_{\varrho'}/f][\tilde{\rho}/\tilde{\varrho}]e\rangle_{[\tilde{\rho}/\tilde{\varrho}]\rho}] \end{array} & (\text{R-RA}\textsc{pp}) \\
\begin{array}{c} E[\textbf{let } f = \textbf{fix}(f, \Lambda\tilde{\varrho}.(\lambda x.e_1 \textbf{ at } \rho)) \textbf{ at } \rho' \textbf{ in } e_2] \\ \longrightarrow E[[\langle \textbf{fix}(f, \Lambda\tilde{\varrho}.(\lambda x.e_1 \textbf{ at } \rho))\rangle_{\rho'}/f]e_2] \end{array} & (\text{R-F}\textsc{ix}) \\
E[\textbf{if0 } \langle 0\rangle_\varrho \textbf{ then } e_1 \textbf{ else } e_2\ ] \longrightarrow E[e_1] & (\text{R-I}\textsc{f}\text{T}) \\
E[\textbf{if0 } \langle n\rangle_\varrho \textbf{ then } e_1 \textbf{ else } e_2\ ] \longrightarrow E[e_2] \qquad (\text{if } n \neq 0) & (\text{R-I}\textsc{f}\text{F}) \\
E[\textbf{letregion } \varrho \textbf{ in } v] \longrightarrow E[[\bullet/\varrho]v] & (\text{R-R}\textsc{eg})
\end{array}
$$

Fig. 1. Reduction rules

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.

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

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 Type Syntax

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

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

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. 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 $[(\mathbf{rec}\ r.\mu_1 \vee \cdots \vee \mu_n)/r]\mu_1, \ldots, [(\mathbf{rec}\ r.\mu_1 \vee \cdots \vee \mu_n)/r]\mu_n$. 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, we require that the outermost type constructors of $\mu_1, \ldots, \mu_n$ are different from each other. For example, $\mathbf{rec}\ r.(\mathbf{num}, \rho) \vee (\mathbf{num}, \rho')$ is invalid. When r does not appear in $\mu_1, \ldots, \mu_n$, we write $\mu_1 \vee \cdots \vee \mu_n$ for $\mathbf{rec}\ r.\mu_1 \vee \cdots \vee \mu_n$.

A type scheme $\forall \tilde{\varrho}^{\varphi} \forall \tilde{\alpha} \forall \tilde{\xi}.\tau$ 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. 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).

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 $\tau' \prec \forall \tilde{\alpha} \forall \tilde{\xi}.\tau$ used in T-RApp and T-VRApp means that there exist $\tilde{\tau''}$ and $\tilde{\varphi}$ such that $\tau' = [\tilde{\tau''}/\tilde{\alpha}][\tilde{\varphi}/\tilde{\xi}]\tau$. 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 $\forall \tilde{\varrho}^{\varphi}.\forall \tilde{\alpha}.\forall \tilde{\xi}.$) appearing in Γ and τ respectively.

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

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

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

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

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

$$\frac{\Gamma\vdash e_1:\tau_1\&\varphi_1 \qquad (\tau_2\xrightarrow{\varphi_0}\tau_3,\rho)\subseteq\tau_1 \qquad \Gamma\vdash e_2:\tau_2\&\varphi_2}{\Gamma\vdash e_1e_2:\tau_3\&\varphi_0\cup\varphi_1\cup\varphi_2\cup\{\rho\}}\quad\text{(T-App)}$$

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

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

Fig. 2. Typing rules for static expressions

Note that in the rule T-App, e_1 need not be a function, since τ may be $(\mathbf{num},\rho')\vee(\tau_2\xrightarrow{\varphi_0}\tau_3,\rho)$. When e_1e_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 $\varphi_2\cup\{\rho\}$ 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_1e_2 is evaluated, irrespectively of whether the value of e_1 is a function or not. (Here, we assume that information about whether a value of type $(\mathbf{num},\rho')\vee(\tau_2\xrightarrow{\varphi_0}\tau_3,\rho)$ is an integer or a function is embedded in the pointer, rather than in the memory cell stored in ρ' or ρ. So, no region is accessed when it is checked whether the value of e_1 is a function or not. If we store that information in the memory cell in ρ' or ρ, we should add all the outermost regions appearing in τ_1 to the effect of e_1e_2 in T-App.)

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

$$\frac{\begin{array}{c}\Gamma \vdash v : (\forall \tilde{\varrho}^{\varphi} \forall \tilde{\alpha} \forall \tilde{\xi}.\tau, \rho_f) \\ \tau' \prec \forall \tilde{\alpha} \forall \tilde{\xi}.[\tilde{\rho'}/\tilde{\varrho}]\tau\end{array}}{\Gamma \vdash v[\tilde{\rho'}] : \tau' \& \{\rho_f\} \cup [\tilde{\rho'}/\tilde{\varrho}]\varphi} \quad \text{(T-VRApp)}$$

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

$$\frac{\begin{array}{c}\Gamma + \{x \mapsto \tau_1\} \vdash e : \tau_2 \& \varphi' \\ \varphi' \subseteq \varphi \\ (\tau_1 \xrightarrow{\varphi} \tau_2, \rho) \subseteq \tau_3\end{array}}{\Gamma \vdash \langle \lambda x.e \rangle_\rho : \tau \& \emptyset} \quad \text{(T-VAbs)}$$

$$\frac{\begin{array}{c}\pi = \forall \tilde{\varrho}^{\varphi} \forall \tilde{\xi}.\tau \\ \{\tilde{\varrho}, \tilde{\xi}, \tilde{\alpha}\} \cap (\mathbf{fv}(\Gamma) \cup \{\rho_f\}) = \emptyset \\ \Gamma + \{f \mapsto (\pi, \rho_f)\} \vdash \lambda x.e_1 \textbf{ at } \rho_t : \tau \& \varphi \\ \pi' = \forall \tilde{\varrho} \forall^{\varphi} \tilde{\alpha} \forall \tilde{\xi}.\tau\end{array}}{\Gamma \vdash \langle \mathbf{fix}(f, \Lambda \tilde{\varrho}.(\lambda x.e \textbf{ at } \rho_t)) \rangle_{\rho_f} : (\pi', \rho_f) \& \emptyset} \quad \text{(T-VFix)}$$

Fig. 3. Typing rules for dynamic expressions

the latter, we obtain

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

We can also obtain

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

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

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

Finally, by using T-Reg, we obtain:

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

4 Type Soundness

The soundness of the type system is guaranteed by Theorems 4.3 and 4.4 given below. Theorem 4.3 implies that a well-typed, closed expression does not access a deallocated region immediately. Theorem 4.4 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.

Remark 4.1: 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 .

Lemma 4.2: If $\emptyset \vdash E : \tau\&\varphi$ is derivable from $\emptyset \vdash [\,] : \tau'\&\varphi'$ and $\bullet \in \varphi'$, then $\bullet \in \varphi$.

Proof: This follows by straightforward induction on derivation of $\emptyset \vdash E : \tau\&\varphi$. $\square$

Theorem 4.3: Suppose $\emptyset \vdash e : \tau\&\varphi$, and e is one of the following forms:

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

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

Proof: We show only the first case. The other cases are similar. Suppose that $\emptyset \vdash E[n \textbf{ at } \rho] : \tau\&\varphi$ and $\bullet \notin \varphi$. By the typing rules, $\emptyset \vdash E[n \textbf{ at } \rho] : \tau\&\varphi$ must have been derived from $\emptyset \vdash n \textbf{ at } \rho : \tau'\&\{\rho\}$. By Lemma 4.2, $\bullet \notin \{\rho\}$, which implies $\rho \neq \bullet$. $\square$

Theorem 4.4 [subject reduction]: If $\Gamma \vdash e : \tau\&\varphi$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau\&\varphi'$ for some φ' such that $\varphi' \subseteq \varphi$.

A proof of this theorem is given in Appendix A.

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. This can be carried out in a way similar to the ordinary region inference [18]. 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 [18].)

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 $(\mathbf{num}, \rho_1) \vee ((\mathbf{num}, \rho_2) \stackrel{\emptyset}{\longrightarrow} (\mathbf{num}, \rho_2), \rho_3)$. 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.

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

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 can 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:

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

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

set! One way to deal with **set!** is to translate **set!** into ML-like operations on reference cells and then perform region inference in the same way as that for ML [19]. To perform the translation, we first perform a whole 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, $(\mathbf{let}\ ((x\ (+\ a\ 1)))\ \ldots\ (\mathbf{set!}\ x\ 2))$ is translated to $(\mathbf{let}\ ((x\ (\mathbf{ref}\ (+\ a\ 1))))\ \ldots\ (:=\ x\ 2))$. Here, $\mathbf{ref}\ v$ 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 interprets the target program. We have not yet implemented a back-end compiler to translate the region-annotated program into machine code, since we need to implement other optimizations [1, 2] to make the region-based memory management more effective. For the experiments reported below, we have inserted instructions for counting memory allocation/deallocation in the interpreter. 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 those programs were translated correctly. (If the translation had been incorrect, the interpreter would have reported a run-time error.) For example, the following program (which takes a binary tree as an input and computes the number of leaves):

```
(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?[r57r62] v2 ))
             (letregion (r67 r69 r88 )
                 (+[r88r67r59r69]
                     (letregion (r73 )
                        (leafcount[r73r57r88r76]
                        (letregion (r82 ) (car[r57r82] v2 )) ))
                     (letregion (r86 )
```

```
                    (leafcount[r86r57r88r89]
                    (letregion (r95 ) (cdr[r57r95] v2 )) )) ))
            1 at r59))
      at r60)
at r52  )
```

Here, `reglambda` creates a region-polymorphic function. The instruction `leafcount[r73r57r88r76]` 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`.

The result of the experiments is summarized in Table 1 and Figure 4. Programs `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.

Table 1 shows the size of each program, the maximum heap size, and the total size of allocated memory cells. In measuring the memory size, we assumed that the size of a number (an integer or a floating point number) is 8 bytes (bignum is not supported), that the size of a function closure is 32 bytes (the effect of the free variables on the size of the closure was ignored), and that the size of a vector is $4 + 4 \times$ (the number of elements). 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 291.6 KBytes, but the required heap space was 17.7 KBytes.

Figure 4 shows the transition of the heap size for each program. We can observe that memory is repeatedly deallocated during execution. For `Tak`, `Div`, `Destruct`, and `RayTracing`, the heap size still grows gradually (but more slowly than without the region-based memory management), so that they will suffer from memory leak for a larger input. We think that this is mainly due to the stack-based management of regions, and can be improved by applying optimizations for the region-based memory management (such as storage mode analysis) [1, 2]. Judging from the result of the above experiments, we think that even without those optimizations, our region inference would be useful for reducing the frequency of garbage collection.

Program Name	Program Size (Lines)	Max. Heap Size (Bytes)	Memory Allocation (Bytes)
`Tak`	23	2748	16488
`Div`	54	9308	43352
`Deriv`	65	1648	2376
`Destruct`	72	1712	23212
`Tree`	10	1780	5080
`RayTracing`	1683	17744	291600

Table 1. Heap size and allocated memory size

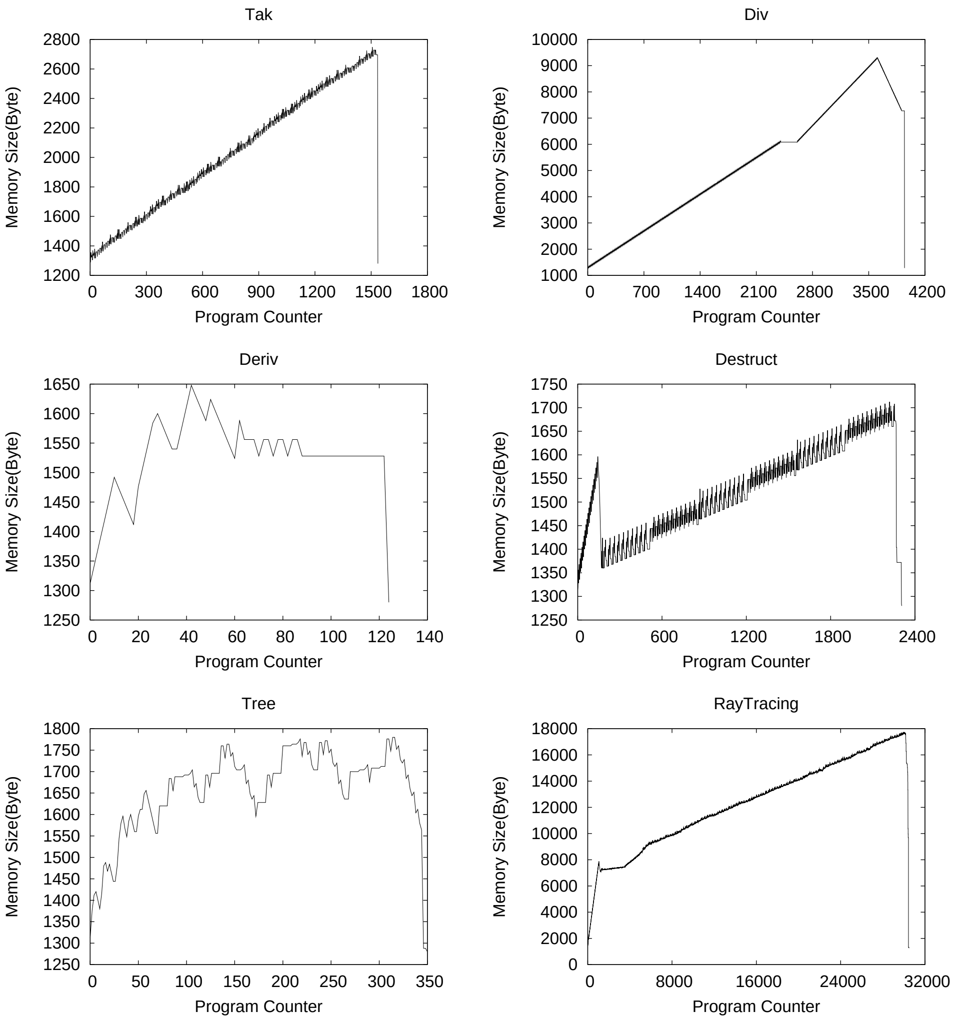

Fig. 4. Transition of the heap size

8 Related Work

Region-based memory management has been applied to programming languages other than ML [3, 6, 7, 9–11, 15, 16], 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 [15, 16] 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.

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].

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 [14, 20], exception analysis [17], and resource usage analysis [12].

References

1. Alexander Aiken, Manuel Fahndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, 1995.
2. Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 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. Cristiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2):199–221, 2002.
5. Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of PLDI'91*, pages 278–292. ACM Press, 1991.
6. Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 262–275, New York, NY, 1999.
7. Morgan Deters and Ron K. Cytron. Automated discovery of scoped memory regions for real-time java. In *Proceedings of ISMM'02*, pages 25–35. ACM Press, 2002.
8. Richard Gabriel. Scheme version of the gabriel lisp benchmarks, 1988.
9. David Gay and Alexander Aiken. Memory management with explicit regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–323, 1998.
10. David Gay and Alexander Aiken. Language support for regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, 2001.
11. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, 2002.
12. Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 331–342, 2002.
13. Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
14. Naoki Kobayashi. Quasi-linear types. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 29–42, 1999.
15. Henning Makholm. Region-based memory management in Prolog. Master's thesis, DIKU, University of Copenhagen, 2000.
16. Henning Makholm. A region-based memory manager for Prolog. In Bart 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.
17. F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 276–290, 1999.
18. Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767, July 1998.
19. Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proceedings of POPL'94*, pages 188–201. ACM Press, January 1994.
20. David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 1–11, San Diego, California, 1995.

Appendix

A A Proof of Theorem 4.4

Lemma A.1 [substitution lemma]: Let σ_1 be τ_1 or (π_1, ρ_1) and σ_2 be τ_2 or (π_2, ρ_2). If $\Gamma + \{x \mapsto \sigma_1\} \vdash e : \sigma_2 \& \varphi$ and $\Gamma \vdash v : \sigma_1 \& \varphi'$, then $\Gamma \vdash [v/x]e : \sigma_2 \& \varphi$.

Proof: By the typing rules, $\varphi' = \emptyset$. Therefore, The derivation for $\Gamma \vdash [v/x]e : \sigma_2 \& \varphi$ can be obtained from the derivation of $\Gamma + \{x \mapsto \sigma_1\} \vdash e : \sigma_2 \& \varphi$ by replacing $\Gamma' + \{x \mapsto \sigma_1\} \vdash x : \sigma_1 \& \emptyset$ with the derivation of $\Gamma' \vdash v : \sigma_1 \& \emptyset$ and replacing $\Gamma' + \{x \mapsto \sigma_1\} \vdash x[\tilde{\rho'}] : \tau_1' \& \varphi''$ with $\Gamma' \vdash v[\tilde{\rho'}] : \tau_1' \& \varphi''$ $\square$

Lemma A.2: If $\Gamma \vdash e : \tau \& \varphi$, then $\theta\Gamma \vdash \theta e : \theta\tau \& \theta\varphi$ for any substitution θ on type, effect, and region variables.

Proof: Straightforward induction on derivation of $\Gamma \vdash e : \tau \& \varphi$. $\square$

Proof of Theorem 4.4: The proof proceeds by case analysis on the rule used for deriving $e \longrightarrow e'$. It is sufficient to show the case for $E = [\,]$ for each rule.

- R-Int: In this case, $e = n$ **at** ϱ and $e' = \langle n \rangle_\varrho$. By the assumption $\Gamma \vdash e : \tau \& \varphi$, it must be the case that $(\mathbf{num}, \rho) \subseteq \tau$ and $\varphi = \{\rho\}$. Let $\varphi' = \emptyset$. Then, we obtain $\Gamma \vdash e' : \tau \& \varphi'$ by using T-VInt.
- R-Abs: Similar to the case for R-Int.
- R-App: In this case, $e = \langle \lambda x.e_1 \rangle_\varrho v$ and $e' = [v/x]e_1$. By the assumption $\Gamma \vdash e : \tau \& \varphi$ and rule T-App, the following conditions must hold:

$$\begin{array}{l} \Gamma \vdash \langle \lambda x.e_1 \rangle_\varrho : \tau_1 \& \varphi_1 \\ (\tau_2 \xrightarrow{\varphi_0} \tau, \rho) \subseteq \tau_1 \\ \Gamma \vdash v : \tau_2 \& \varphi_2 \\ \varphi = \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\rho\} \end{array}$$

  The first condition must have been derived by using T-VAbs, so that the following conditions must also hold:

$$\begin{array}{l} \Gamma + \{x \mapsto \tau_2\} \vdash e_1 : \tau \& \varphi_0' \\ \varphi_0' \subseteq \varphi_0 \end{array}$$

  By the substitution lemma (Lemma A.1), we have $\Gamma \vdash [v/x]e_1 : \tau \& \varphi_0'$. Moreover, we have $\varphi_0' \subseteq \varphi_0 \subseteq \varphi$ as required.
- R-RApp: In this case, $e = v[\tilde{\rho}]$ and $e' = \langle \lambda x.[v/f][\tilde{\rho}/\tilde{\varrho}]e_1 \rangle_{[\tilde{\rho}/\tilde{\varrho}]\rho''}$ where $v = \langle \mathbf{fix}(f, \Lambda\tilde{\varrho}.(\lambda x.e_1 \ \mathbf{at} \ \rho'')) \rangle_{\rho'}$. By the assumption $\Gamma \vdash e : \tau \& \varphi$, the following conditions must hold:

$$\begin{array}{l} \Gamma \vdash v : (\pi', \rho') \& \emptyset \\ \pi' = \forall \tilde{\varrho}^{\varphi_1} \forall \tilde{\alpha} \forall \tilde{\epsilon}.\tau_1 \\ \tau \prec \forall \tilde{\alpha} \forall \tilde{\epsilon}.[\tilde{\rho}/\tilde{\varrho}]\tau_1 \\ \varphi = \{\rho'\} \cup [\tilde{\rho}/\tilde{\varrho}]\varphi_1 \\ \pi = \forall \tilde{\varrho}^{\varphi_1} \forall \tilde{\epsilon}.\tau_1 \\ \{\tilde{\varrho}, \tilde{\epsilon}, \tilde{\alpha}\} \cap (\mathbf{fv}(\Gamma) \cup \{\rho'\}) = \emptyset \\ \Gamma + \{f \mapsto (\pi, \rho')\} \vdash \lambda x.e_1 \ \mathbf{at} \ \rho'' : \tau_1 \& \varphi_1 \end{array}$$

From the last condition, we obtain $\Gamma + \{f \mapsto (\pi, \rho')\} \vdash \langle \lambda x.e_1 \rangle_{\rho''} : \tau_1 \& \emptyset$. By $\Gamma \vdash v : (\pi', \rho') \& \emptyset$ and rule T-VFIX, we also have $\Gamma \vdash v : (\pi, \rho') \& \emptyset$. By applying Lemma A.1, we obtain:

$$\Gamma \vdash [v/f]\langle \lambda x.e_1 \rangle_{\rho''} : \tau_1 \& \emptyset.$$

By applying Lemma A.2, we further obtain

$$[\tilde{\rho}/\tilde{\varrho}]\Gamma \vdash [\tilde{\rho}/\tilde{\varrho}]([v/f]\langle \lambda x.e_1 \rangle_{\rho''}) : [\tilde{\rho}/\tilde{\varrho}]\tau_1 \& \emptyset.$$

Since $e' = [\tilde{\rho}/\tilde{\varrho}]([v/f]\langle \lambda x.e_1 \rangle_{\rho''})$ and $[\tilde{\rho}/\tilde{\varrho}]\Gamma = \Gamma$, we have

$$\Gamma \vdash e' : [\tilde{\rho}/\tilde{\varrho}]\tau_1 \& \emptyset.$$

By Lemma A.2 and the conditions $\tau \prec \forall \tilde{\alpha} \forall \tilde{\epsilon}.[\tilde{\rho}/\tilde{\varrho}]\tau_1$ and $\{\tilde{\varrho}, \tilde{\epsilon}, \tilde{\alpha}\} \cap \mathbf{fv}(\Gamma) = \emptyset$, we have $\Gamma \vdash e' : \tau \& \emptyset$ as required.

– R-FIX: In this case, $e = \mathbf{let}\ f = \mathbf{fix}(f, \Lambda\tilde{\varrho}.(\lambda x.e_1\ \mathbf{at}\ \rho))\ \mathbf{at}\ \rho'\ \mathbf{in}\ e_2$ and $e' = [\langle \mathbf{fix}(f, \Lambda\tilde{\varrho}.(\lambda x.e_1\ \mathbf{at}\ \rho))\rangle_{\rho'}/f]e_2$. By the assumption $\Gamma \vdash e : \tau \& \varphi$, the following conditions must hold:

$$\begin{array}{l}
\pi = \forall \tilde{\varrho}^{\varphi_1} \forall \tilde{\epsilon}.\tau_1 \\
\{\tilde{\varrho}, \tilde{\epsilon}, \tilde{\alpha}\} \cap (\mathbf{fv}(\Gamma) \cup \{\rho'\}) = \emptyset \\
\Gamma + \{f \mapsto (\pi, \rho')\} \vdash \lambda x.e_1\ \mathbf{at}\ \rho : \tau_1 \& \varphi_1 \\
\pi' = \forall \tilde{\varrho}^{\varphi_1} \forall \tilde{\alpha} \forall \tilde{\epsilon}.\tau_1 \\
\Gamma + \{f \mapsto (\pi', \rho')\} \vdash e_2 : \tau \& \varphi_2 \\
\varphi = \{\rho'\} \cup \varphi_2
\end{array}$$

From the first four conditions, we obtain $\Gamma \vdash \langle \mathbf{fix}(f, \Lambda\tilde{\varrho}.(\lambda x.e_1\ \mathbf{at}\ \rho))\rangle_{\rho'} : (\pi', \rho') \& \emptyset$. By Lemma A.1, we have $\Gamma \vdash e' : \tau \& \varphi_2$. Moreover, we have $\varphi_2 \subseteq \varphi$ as required.

– R-IFT: In this case, $e = \mathbf{if0}\ \langle 0 \rangle_{\varrho}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$ and $e' = e_1$. By the assumption $\Gamma \vdash e : \tau \& \varphi$, there must exist φ' such that $\Gamma \vdash e_1 : \tau \& \varphi'$ and $\varphi' \subseteq \varphi$.

– R-IFF: Similar to the case for R-IFT.

– R-REG: In this case, $e = \mathbf{letregion}\ \varrho\ \mathbf{in}\ v$ and $e' = [\bullet/\varrho]v$. By the assumption $\Gamma \vdash e : \tau \& \varphi$, we have:

$$\begin{array}{l}
\Gamma \vdash v : \tau \& \varphi' \\
\varphi = \varphi' \setminus \{\varrho\} \\
\varrho \notin \mathbf{fv}(\Gamma, \tau)
\end{array}$$

By the typing rules for values, $\varphi' = \emptyset$. By applying Lemma A.2, we obtain $\Gamma \vdash [\bullet/\varrho]v : \tau \& \emptyset$.

$\square$