Region Type Checking for Core-Java

Wei-Ngan Chin[1,3], Shengchao Qin[1,3], and Martin Rinard[2,4]
[1]Singapore-MIT Alliance, E4-04-10, 4 Engineering Drive 3, Singapore 117576
[2]Singapore-MIT Alliance, Building 8-407, 77 Massachusetts Avenue, Cambridge, MA 02139 USA
[3]Department of Computer Science, National University of Singapore
[4]Laboratory for Computer Science, Massachusetts Institute of Technology

***Abstract*— Region-based memory management offers several important advantages over garbage-collected heap, including real-time performance, better data locality and efficient use of limited memory. The concept of regions was first introduced for a call-by-value functional language by Tofte and Talpin, and has since been advocated for imperative and object-oriented languages. Scope memory, a lexical variant of regions, is now a core feature in a recent proposal on Real-Time Specification for Java (RTSJ). In this paper, we propose a region-based memory management system for a core subset of Java. Our region type analysis can completely prevent dangling references and thus is ready to cater for the no-dangling requirement in RTSJ. Our system also supports modular compilation, which is an important feature for Java, but was missing in recent related work.**

***Index Terms*— Core-Java, Region Type, Type Checking.**

I. Introduction

In region-based memory management, each new object is added to a *region* with a designated lifetime. While objects may be added to their respective regions at different times, the entire set of objects in each region are freed simultaneously, when the region is deleted. Various studies have shown that region-based programming can provide safe memory management with good real-time performance. Data locality is also improved when related objects are placed together in the same region. By classifying objects into regions based on their lifetimes, better utilization of memory can be achieved when dead regions are recovered on a timely basis.

Region-based memory management was first invented in 1994 by Tofte and Talpin[14], and has been implemented for Standard ML[13]. A type and effect program analysis was used to associate every value with a region, together with automatic inference on the lifetime (or scope) of these regions. Region type rules guarantee that a well-typed region-annotated program never access a dangling reference to a region that has already been deallocated. Recently, several works have investigated region-based programming for Java-based languages [9], [5], [4]. Most of these works (e.g. [5], [4]) use region type-checking to guarantee that well-typed programs never access dangling references, so that runtime tests for dangling references can be omitted.

Our main contribution is a new region type system, which has the following important features.

- Our region type rules prevent dangling references by requiring that all field references outlive the current object. We formalise this explicitly through region lifetime constraints, without the need for an effect-based typing.
- In comparison with the work [5], our system adopts the open world assumption for region annotations, thus modular compilation is possible.
- Our system supports classes and methods with region-polymorphism, and also supports polymorphic recursion for methods.

The remainder of this paper is organised as follows. Section II introduces the language and the region parameterized type. We present the region type rules in Section III. Section IV discusses related works, followed by some concluding remarks in Section V.

II. Core-Java and Region Types

In this section, we present the syntax of a significant (subset of) Java-like language named *Core-Java* and introduce region types we shall adopt.

Fig 1 presents the syntax for our language Core-Java. Core-Java is designed in the same minimalist spirit as Featherweight Java[12]. It supports assignments but is also an expression-oriented language where statements are expressions with the void type. There is no while loop construct as this can be captured by tail-recursion. The language is extended with region types and region constraints for each class and method. Also, local region with a designated scope is introduced by the **letreg** declaration.

Note that r denotes a region variable, while v represents a data variable. The suffix notation s^* denotes a list of zero or more distinct syntactic terms that are separated by appropriate separators, while s^+ represents a list of one or more distinct syntactic terms. The syntactic terms, s, could be v, r, $(t\ v)$, *field*, etc. For example, $(t\ v)^*$ denotes $(t_1\ v_1, .., t_n\ v_n)$ where $n \geq 0$.

The constraint $r_1 \succeq r_2$ indicates that the lifetime of region r_1 is not shorter than that of r_2. The constraint $r_1 = r_2$ denotes that r_1 and r_2 must be the same region, while $r_1 \neq r_2$ denotes the converse. Note that *this* is a reserved data variable referring to the current object, while *heap* is a reserved region

Wei-Ngan Chin is with the Department of Computer Science, School of Computing, National University of Singapore,3 Science Drive 2, Singapore 117543, Republic of Singapore. Email: chinwn@comp.nus.edu.sg

Shengchao Qin is the author for correspondence. He is with the Department of Computer Science, School of Computing, National University of Singapore,3 Science Drive 2, Singapore 117543, Republic of Singapore. Email: qinsc@comp.nus.edu.sg. Tel: +65-6874 1298

Martin Rinard is with the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, 545 Technology Square NE43-620A Cambridge, MA 02139. Email: rinard@lcs.mit.edu

$$
\begin{array}{rcl}
P & ::= & \mathit{def}^*\ \mathit{meth}^*\ eb \\
\mathit{def} & ::= & \textbf{class}\ ca_1\ \textbf{extends}\ ca_2\ \textbf{where}\ rc\ \{\mathit{field}^*\ \mathit{meth}^*\} \\
ca & ::= & cn\langle r^+\rangle \\
\mathit{field} & ::= & t\ f \\
\mathit{meth} & ::= & t\ mn\langle r^*\rangle\textbf{((}t\ v\textbf{)}^*\textbf{)}\ \textbf{where}\ rc\ eb \\
t & ::= & ca\ \mid\ \mathit{prim}\langle\rangle \\
\mathit{prim} & ::= & \textbf{int}\ \mid\ \textbf{bool}\ \mid\ \textbf{void} \\
eb & ::= & \{(t\ v)^*\ \ e\} \\
e & ::= & \textbf{(}ca\textbf{) null}\ \mid\ k\ \mid\ v\ \mid\ v.f\ \mid\ eb \\
 & & \mid\ \textbf{new}\ ca\textbf{(}v^*\textbf{)}\ \mid\ v.f = e\ \mid\ v = e \\
 & & \mid\ v.mn\langle r^*\rangle\textbf{(}v^*\textbf{)}\ \mid\ mn\langle r^*\rangle\textbf{(}v^*\textbf{)}\ \mid\ e_1\ \textbf{;}\ e_2 \\
 & & \mid\ \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2\ \mid\ \textbf{letreg}\ r\ \textbf{in}\ eb \\
rc & ::= & r_1 \succeq r_2\ \mid\ r_1 = r_2\ \mid\ r_1 \neq r_2\ \mid \\
 & & \mid\ rc_1 \wedge rc_2\ \mid\ \mathit{true}
\end{array}
$$

Fig. 1. The Syntax of Core-Java

```
class Pair⟨r1,r2,r3⟩ extends Object⟨r1⟩
  where r2≥r1 ∧ r3≥r1 {
  Object⟨r2⟩ fst
  Object⟨r3⟩ snd
  Object⟨r4⟩ getFst⟨r4⟩()
   where r2=r4 {fst}
  void setSnd⟨r4⟩(Object⟨r4⟩ o)
   where r3=r4 {snd=o}
  Pair⟨r4,r5,r6⟩ cloneRev⟨r4,r5,r6⟩()
    where r2=r6 ∧ r3=r5 {
    Pair⟨r4,r5,r6⟩ tmp
       =new Pair⟨r4,r5,r6⟩(null,null);
    tmp.fst=snd; tmp.snd=fst;
    tmp
  }
  void swap() where r2=r3 {
    Object⟨r2⟩ tmp=fst;
    fst=snd;
    snd=tmp;
  }
}
```

Fig. 2. The Class `Pair`

to denote the global heap with unlimited lifetime, that is for any r: $\mathit{heap} \succeq r$.

Each class definition is parameterised with one or more regions, to form a region type [14], [15], [5], [4]. For instance, $cn\langle r_1, ..., r_n\rangle$ is a class annotated with region parameters $r_1, ..., r_n$. Parameterisation allows programmers to implement a region-polymorphic class whose components can be allocated in different regions. The first region parameter r_1 is special: it refers to the region in which a specified object of this class will be allocated. All other regions should *outlive* this region via the constraint $\wedge_{i:1..n}(r_i \succeq r_1)$. That is the regions of the components (possibly in $r_1, ..., r_n$) should have longer or equal lifetimes than the region (namely r_1) of its object. This condition, called *no-dangling* requirement, can prevent dangling references completely, as it guarantees that each object can never reference another object in a younger region.

Every method is also decorated with zero or more region parameters. They are intended to capture the regions used by each method's parameters and result. Each method also has a region lifetime constraint that is consistent with the operations performed in the method body.

Two example programs are shown in Fig 2 and Fig 3 for the `Pair` and `PList` classes.

Note that in Fig. 2 the regions (`r1,r2,r3`) for the subclass (`Pair`) is an extension of that (`r1`) of the superclass (`Object`). The class invariant (`r2≥r1 ∧ r3≥r1`) ensures that the regions for the components outlive the region for the object. Also notice that each method (e.g. `cloneRev`) has a pre-condition (e.g. `r2≥r6 ∧ r3≥r5`) that should be satisfied at call sites before the method is invoked. With these constraints being satisfied when instantiated with actual regions, no dangling references can occur during the execution of the program.

The `PList` class in Fig. 3 describes a recursive list, elements of which are of type `Pair`. For simplicity, and also for data locality, we adopts region-monomorphism for recursive classes. Therefore, the recursive field has the same region annotation as its class.

```
class PList⟨r1,r2,r3,r4⟩ extends Object⟨r1⟩
  where r2≥r1∧r3≥r1∧r4≥r1∧r3≥r2∧r4≥r2 {
  Pair⟨r2,r3,r4⟩ p
  PList⟨r1,r2,r3,r4⟩ next
  Pair⟨r5,r6,r7⟩ getPair⟨r5,r6,r7⟩()
   where r5=r2∧r6=r3∧r7=r4
    {p}
  PList⟨r5,r6,r7,r8⟩ getNext⟨r5,r6,r7,r8⟩()
   where r5=r1∧r6=r2∧r7≥r3∧r8=r4
    {next}
  void setNext⟨r5,r6,r7,r8⟩
   (PList⟨r5,r6,r7,r8⟩ o)
   where r5=r1∧r6=r2∧r7≥r3∧r8=r4
    {next = o}
}
```

Fig. 3. The Class `PList`

III. Region Type System

In this section we build a region type system for Core-Java. We shall formulate the region type rules for Core-Java through the following set of typing relations.

- $\mathit{def} \in P$. It denotes a class within a given program P.
- $\mathit{meth} \in P$. It denotes a static method within a given program P.
- $P \vdash \mathit{mbr} \in \mathit{ca}$. It indicates that *mbr* is a member of class *ca*, either declared in current class *ca*, or inherited from its superclass. *mbr* can be a field or a method. Direct membership without inheritance is captured by $P \vdash \mathit{mbr} \in_D \mathit{ca}$.

- $P; R \vdash_{constr} t,\ \varphi$. The constraint that should be imposed on type t is φ.
- $P; R; \Psi \vdash_{type} t$. The type t is well-formed with respect to program P, the set of alive regions R and the region constraint Ψ.
- $P; R; \Psi \vdash_{formal} t\ v$. A field ($t\ v$) in a class or a formal parameter of a method is well formed.
- $P \vdash cn\langle x_{1..m}\rangle <: cn\langle x'_{1..p}\rangle, \rho$. This captures a subtype relation between $cn\langle x_{1..q}\rangle$ and $cn'\langle x'_{1..q}\rangle$ with substitution ρ to enforce identical parameter regions.
- $P; R; \Psi \vdash t <: t'$. This states that t is a subtype of t' under the constraint Ψ.
- $P \vdash_{def} def$. It denotes that class declaration *cdecl* is well-formed in program P.
- $P; \Gamma; R; \Psi \vdash_{meth} meth$. It specifies that method *meth* is well-defined with respect to program P, type environment Γ, live regions R and the region constraint Ψ.
- $P; \Gamma; R; \Psi \vdash e : t$. It indicates that expression e is of type t with respect to program P, type environment Γ, the set of alive regions R and the constraint Ψ.
- $\vdash P : t$. It indicates that program P is well-formed with main expression of type t.

We shall begin with some auxiliary typing relations that are used to extract classes, methods and fields from a given program, together with checks on the well-formedness of types, parameters and subtyping.

- The following two rules are for extracting the classes and static methods from a given program.

$$\frac{P = ...def...\ meth^*\ eb\ Q}{def \in P}$$

$$\frac{P = def^*\ ...meth...\ eb\ Q}{meth \in P}$$

- Methods/fields defined in a class.
  Let $mbr = field \mid meth$ be a class member (i.e. field or instance method). We first define direct membership using the following rules:

$$\frac{\textbf{class}\ cn\langle r_{1..n}\rangle ...\{...mbr...\}\ \in\ P}{P \vdash mbr \in_D cn\langle r_{1..n}\rangle}$$

  Fields/methods may be inherited through the superclass. This could be dealt by the following membership with inheritance relation.

$$\frac{P \vdash mbr \in_D cn\langle r_{1..n}\rangle}{P \vdash mbr \in cn\langle r_{1..n}\rangle}$$

$$\frac{\begin{array}{c}\textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ cn'\langle r_{1..m}\rangle ...\ \in\ P \\ P \vdash mbr \in cn'\langle r_{1..m}\rangle \quad P \vdash mbr \notin_D cn\langle r_{1..n}\rangle\end{array}}{P \vdash mbr \in cn\langle r_{1..n}\rangle}$$

  For convenience, we may drop P from our relations using $mbr \in cn\langle r_{1..n}\rangle$.
- Well-formedness of types and fields/parameters.
  The first three rules derive the invariant associated with each type. For each class type, there is some invariant describing the lifetime relation on regions involved, while for primitive and *Object* types, no invariant is imposed since they have at most one region parameter.

$$\frac{}{P; R \vdash_{constr} prim\langle\rangle,\ true}$$

$$\frac{r \in R}{P; R \vdash_{constr} Object\langle r\rangle,\ true}$$

$$\frac{\begin{array}{c}def = \textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ c\ \textbf{where}\ \varphi\ \{...\} \\ def \in P \quad R \supseteq \{x_1, ..., x_n\}\end{array}}{P; R \vdash_{constr} cn\langle x_{1..n}\rangle, [r_1 \mapsto x_1, ..., r_n \mapsto x_n]\varphi}$$

  The next two rules are used to check the well-formedness of types and fields, under a region constraint Ψ.

$$\frac{P; R \vdash_{constr} t,\ \varphi \qquad \Psi \Rightarrow \varphi}{P; R; \Psi \vdash_{type} t}$$

$$\frac{P; R; \Psi \vdash_{type} t}{P; R; \Psi \vdash_{formal} t\ v}$$

- Subtyping relation.
  As mutations are possible on the fields, the subtype relation requires the region parameters to be invariant (both contra-variant and co-variant). This is achieved through equality constraints on the regions of both the subtype and its supertype, as follows.

$$\frac{\rho = [x_i \mapsto x'_i]_{i \in 1..n}}{P \vdash cn\langle x_{1..n}\rangle <: cn\langle x'_{1..n}\rangle, \rho}$$

$$\frac{\begin{array}{c}def = \textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ cn'\langle r_{1..m}\rangle ... \\ def\ \in\ P \\ P \vdash cn'\langle x_{1..m}\rangle <: cn''\langle x'_{1..p}\rangle,\ \rho\end{array}}{P \vdash cn\langle x_{1..n}\rangle <: cn''\langle x'_{1..p}\rangle,\ \rho}$$

$$\frac{\begin{array}{cc}P \vdash t <: t', \rho & \Psi \Rightarrow ctr(\rho) \\ P; R; \Psi \vdash_{type} t & P; R; \Psi \vdash_{type} t'\end{array}}{P; R; \Psi \vdash t <: t'}$$

  The first two rules attempt to derive the needed region equality constraint in terms of a substitution mapping. The third rule checks that the derived equality constraint is valid under Ψ. Note that function $ctr(\rho)$ converts a substitution mapping to its corresponding set of equality constraints. For example, $ctr([x_1 \mapsto v, x_2 \mapsto w]) = (x_1 = v \wedge x_2 = w)$.

We shall now describe typing relations for classes and methods.

- Well formed class definitions: $P \vdash_{def} def$
  This relation states that class declaration *def* is well-formed in program P.

$$\frac{\begin{array}{c} def = \textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ c\ \textbf{where}\ \varphi\ \{field_{1..p}\ meth_{1..q}\} \\ def \in P \qquad \varphi \Rightarrow \forall i : 2..n \cdot r_i \succeq r_1 \\ R = \{r_1, \ldots, r_n, heap\} \\ \forall i : 1..q \cdot P; \{this : cn\langle r_{1..n}\rangle\}; R; \varphi \vdash_{meth} meth_i \\ \forall i : 1..p \cdot P; R; \varphi \vdash_{formal} field_i \end{array}}{P \vdash_{def} def}$$

A class is well defined if all its fields and methods are well-formed and the constraint specifies necessary lifetime relations among its regions.

- Well formed methods: $P; \Gamma; R; \Psi \vdash_{meth} meth$
  This relation specifies that method *meth* is well-defined with respect to program *P*, type environment Γ, the set of live regions *R* and the region constraint Ψ.

$$\frac{\begin{array}{c} \Gamma' = \Gamma, (v_j : t_j)_{j:1..p} \qquad R' = R \cup \{r_1, \ldots, r_m\} \\ P; R'; \Psi' \vdash_{type} t_j,\ j = 0, .., p \qquad \Psi' = \Psi \wedge \varphi_2 \\ P; \Gamma'; R'; \Psi' \vdash eb : t_0' \qquad P; R'; \Psi' \vdash t_0' <: t_0 \end{array}}{P; \Gamma; R; \Psi \vdash_{meth} t_0\ mn\langle r_{1..m}\rangle((t_j\ v_j)_{j:1..p})\ \textbf{where}\ \varphi_2\ eb}$$

Region constraint of variables are checked at declaration block. For each type, all regions used must be alive in *R* and has an invariant that is satisfied by the expected context, Ψ'.

Our next typing relation $P; \Gamma; R; \Psi \vdash e : t$ indicates that expression e is of type t with respect to program *P*, type environment Γ, the set of alive regions *R* and the constraint Ψ. Some cases are defined next.

- Constants and variables.

$$\frac{P; R; \Psi \vdash_{type} cn\langle r_{1..n}\rangle}{P; \Gamma; R; \Psi \vdash (cn\langle r_{1..n}\rangle)\ \textbf{null} : cn\langle r_{1..n}\rangle}$$

$$\frac{P; R; \Psi \vdash_{type} t_k}{P; \Gamma; R; \Psi \vdash k : t_k}$$

$$\frac{(v : t) \in \Gamma}{P; \Gamma; R; \Psi \vdash v : t}$$

Take note that region constraints of variables are checked at declaration sites, rather than at their uses.

- Expression block.

$$\frac{\begin{array}{c} P; R; \Psi \vdash_{type} t_i \quad i = 1, .., p \\ \Gamma' = \Gamma + (v_i : t_i)_{i:1..p} \\ P; \Gamma'; R; \Psi \vdash e : t \end{array}}{P; \Gamma; R; \Psi \vdash \{(t_i\ v_i)_{i:1..p}\ \ e\} : t}$$

- Object creation.

$$\frac{\begin{array}{c} P; R; \Psi \vdash_{type} cn\langle x_{1..n}\rangle \\ fieldlist(cn\langle x_{1..n}\rangle) = (t_i\ f_i)_{i:1..p} \\ (v_i : t_i') \in \Gamma \quad P; R; \Psi \vdash t_i' <: t_i \quad i = 1, .., p \end{array}}{P; \Gamma; R; \Psi \vdash \textbf{new}\ cn\langle x_{1..n}\rangle(v_1, .., v_p) : cn\langle x_{1..n}\rangle}$$

Note that function $fieldlist(cn\langle x_{1..n}\rangle)$ returns a list comprising all available fields in class $cn\langle x_{1..n}\rangle$, including the fields of its superclasses. They are organised in an order determined by the constructor function.

- Object field access.

$$\frac{\begin{array}{c} P \vdash (t\ f) \in cn\langle r_{1..n}\rangle \\ (v : cn\langle a_{1..n}\rangle) \in \Gamma \\ t' = [r_1 \mapsto a_1, ..., r_n \mapsto a_n]t \end{array}}{P; \Gamma; R; \Psi \vdash v.f : t'}$$

- Object field update and assignment.
  Let $lhs = v \mid v.f$ be either a variable v or an object field $v.f$. The following rule covers both object field update and general assignment.

$$\frac{\begin{array}{c} P; \Gamma; R; \Psi \vdash lhs : t \\ P; \Gamma; R; \Psi \vdash e : t' \\ P; R; \Psi \vdash t' <: t \end{array}}{P; \Gamma; R; \Psi \vdash lhs = e : \textbf{void}}$$

- Method invocation.
  We highlight instance method invocation. The rule for static method invocation is similar.

$$\frac{\begin{array}{c} (v_0 : cn\langle a^+\rangle) \in \Gamma \quad P \vdash meth \in cn\langle r^+\rangle \\ meth = t\ mn\langle r'^*\rangle((t_i\ v_i)_{i:1..n})\ \textbf{where}\ \varphi_0\ eb \\ (v_i' : t_i') \in \Gamma,\ i = 1, .., n \\ \rho =_{df} [r^+ \mapsto a^+,\ r'^* \mapsto a'^*] \qquad \Psi \Rightarrow \rho\varphi_0 \\ P; R; \Psi \vdash t_i' <: \rho t_i,\ i = 1, .., n \end{array}}{P; \Gamma; R; \Psi \vdash v_0.mn\langle a'^*\rangle(v'_{1..n}) : \rho t}$$

- Sequential composition.

$$\frac{\begin{array}{c} P; \Gamma; R; \Psi \vdash e_1 : t_1 \\ P; \Gamma; R; \Psi \vdash e_2 : t_2 \end{array}}{P; \Gamma; R; \Psi \vdash e_1\ ;\ e_2 : t_2}$$

- Conditional.

$$\frac{\begin{array}{c} v : \textbf{bool}\langle\rangle \in \Gamma \\ P; \Gamma; R; \Psi \vdash eb_1 : t_1 \quad P; R; \Psi \vdash t_1 <: t \\ P; \Gamma; R; \Psi \vdash eb_2 : t_2 \quad P; R; \Psi \vdash t_2 <: t \end{array}}{P; \Gamma; R; \Psi \vdash \textbf{if}\ v\ \textbf{then}\ eb_1\ \textbf{else}\ eb_2 : t}$$

Conditional branches are checked separately before a most specific super-class is identified through subtyping.

- Region declarations.

$$\frac{\begin{array}{c} \varphi = (R \succ r) \quad freereg(t) \subseteq R \\ P; \Gamma; R \cup \{r\}; \Psi \wedge \varphi \vdash eb : t \end{array}}{P; \Gamma; R; \Psi \vdash \textbf{letreg}\ r\ \textbf{in}\ eb : t}$$

Local blocks are checked with an extra live region, and the constraint $R \succ r$, which is an abbreviation for $\bigwedge_{r' \in R}(r' \succeq r \wedge r' \neq r)$. Also, $freereg(t)$ returns all free region variables in *t*.

Lastly, we give the typing relation for the entire program using $\vdash P : t$. It indicates that program *P* is well-typed (with type *t*).

$$\frac{\begin{array}{c} WFClasses(P) \quad FieldsOnce(P) \\ MethodsOnce(P) \quad InheritanceOK(P) \\ P = def_{1..n}\ meth_{1..m}\ eb\ Q \qquad \forall i : 1..n \cdot P \vdash_{def} def_i \\ \forall i : 1..m \cdot P; \emptyset; \{heap\}; true \vdash_{meth} meth_i \\ P; \emptyset; \{heap\}; true \vdash eb : t \end{array}}{\vdash P : t}$$

A region-annotated program is well typed if all declared classes and static methods are well-formed and the body of the program is well-typed. The predicates in the premise are used to capture standard well-formedness conditions for object-oriented programs. That is:

- no duplicate definitions of classes and no cycle in class hierarchy.
- no duplicate definitions of fields.
- no duplicate definitions of methods.
- soundness of class subtyping and method overriding.

The auxiliary rules for these predicates are omitted here.

Our region type system enjoys several properties, like a well-typed program will never go wrong, the execution of a well-type program will not generate any dangling pointers. These memory-safety properties are very important for region-based memory management. The dynamic semantics and the proofs for all these safety properties will appear in a future paper.

IV. Related Work

Tofte and Talpin [14], [15] proposed a region inference approach for a typed call-by-value λ-calculus, and tested their approach in a region-based implementation of Standard ML. In their approach, all values (including function values) are put into regions at runtime, and all points of region placement can be inferred automatically using a type and effect based program analysis. The treatment of reference type is similar to that for objects; as all values stored in a given reference must be placed in the same region which in turn outlives the region where the reference is located. Apart from this imperative feature, their approach is focused mainly on functional language features.

Christiansen and Velschow proposed a similar region-based approach to memory management in Java [5]. They call their system RegJava, in which a stack of lexically scoped regions are introduced for the allocation of objects. They proposed a region type system and demonstrated its soundness by linking the static semantics with the dynamic semantics. However, their system adopts the close world assumption, thus modular compilation cannot be supported.

Gay and Aiken implemented a region-based extension of C, called C@, which used reference counting on regions to safely allocate and deallocate regions with a minimum of overhead [10]. Using special region pointers and explicit deleteregion calls, they provided a means of explicitly manipulating region-allocated memory. This approach allows non-lexical regions where earlier deallocation of regions are possible, but stack implementation of regions is no longer valid. Their work indicated that region-based programming often use less memory and is faster than traditional malloc/free-based memory management. However, counting escaping references can incur noticeable overhead.

Beebee and Rinard [1] gave an early implementation of scoped memory for Real-Time Java in the MIT Flex compiler infrastructure. They relied on both static analysis and dynamic debugging to help locate incorrect uses of scoped memory. Later, Boyapati *et. al.* [4] combined region types [14], [15], [8], [11], [5] and ownership types [7], [6], [2], [3] in a unified framework to capture object encapsulation and yet prevent dangling references. Their static type system guaranteed that scope-memory runtime checks will never fail for well-typed programs. It also ensured that real-time threads do not interfere with the garbage collector. Using object encapsulation, an object and all components it *owns* are put into the same region; in order to optimize on the regions used. Our region type system is quite similar to theirs, but we separate out object encapsulation and RTSJ issues, and prefer to infer region types automatically (will appear in a future paper).

V. Concluding Remarks

Our eventual aim is to provide a fully-automatic region inference type system for a core subset of Java. To achieve this, we proposed a new region type system in this paper, which allows classes and methods to be region-polymorphic, with polymorphic recursion for methods. We have seen how appropriate region instantiations are decided by the region lifetime constraints, which are meant exclusively at preventing dangling references.

The present work is at a preliminary stage, there remain a number of important issues that should be resolved and improved further. We are currently working on the safety of our region type system and an automatic inference algorithm to insert region annotations into normal Java programs. These results will be reported in a future paper.

Acknowledgments

The authors would like to thank William Beebee, Chandrasekar Boyapati, Mong Leng Sin, Siau-Cheng Khoo, Florin Cracium, Dana Xu, and Nguyen Huu Hai for helpful discussions and various pointers. We would also like to acknowledge the financial support of Singapore-MIT Alliance and research grant R252-000-092-112.

References

[1] W. Beebee and M. Rinard. An Implementation of Scoped Memory for Real-Time Java. In *Proceedings of Embedded Software, First International Workshop (EMSOFT '01)*, Tahoe City, California, October 2001.

[2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '02)*, Seattle, Washington, November 2002.

[3] C. Boyapati, B. Liskov, and L. Shrira. Ownership Types for Object Encapsulation. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, Louisiana, January 2003.

[4] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In *Proceedings of the ACM Conference on Program Language Design and Implementation (PLDI '03)*, San Diego, California, June 2003.

[5] M. V. Christiansen and P. Velschow. Region-Based Memory Management in Java. Master's Thesis, Department of Computer Science (DIKU), University of Copenhagen, 1998.

[6] D. G. Clarke and S. Drossopoulou. Ownership, Encapsulation and Disjointness of Type and Effect. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '02)*, Seattle, Washington, November 2002.

[7] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, October 1998.

[8] K. Crary, D. Walker, and G. Morrisett. Typed Memory Management in a Calculus of Capabilities. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '99)*, January 1999.

[9] M. Deters and R. Cytron. Automated Discovery of Scoped Memory Regions for Real-Time Java. In *Proceedings of the International Symposium on Memory Management (ISMM '02)*, June 2002.

[10] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI '98)*, June 1998.

[11] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI '01)*, June 2001.

[12] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, Denver, Colorado, November 1999.

[13] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T.H. Olesen, and P. Sestoft. *Programming with Regions in the ML Kit (for Version 4)*. The IT University of Copenhagen, September 2001.

[14] M. Tofte and J. Talpin. Implementing the Call-By-Value λ-calculus Using a Stack of Regions. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '94)*, January 1994.

[15] M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.

Wei Ngan Chin is an Associate Professor at the Department of Computer Science, School of Computing, National University of Singapore, and a Fellow in the Computer Science Programme of the Singapore-MIT Alliance. He received his B.Sc. and M.Sc. in Computer Science, in 1982 and 1983, respectively, from University of Manchester, United Kingdom, and PhD in Computing, in 1990 from the Imperial College of Science, Technology and Medicine, United Kingdom. His current research interests are functional programming, program transformation, parallel systems, software models and methods.

Shengchao Qin is a research fellow in the National University of Singapore under the Singapore-MIT Alliance (SMA). He received his BSc in information science and PhD in applied mathematics, in 1997 and 2002, respectively, from Peking University, China. His main research interests include type-based program analysis, formal specification and verification, refinement calculus and transformation, formal techniques in hardware/software co-design, embedded systems.

Martin Rinard is an Associate Professor in the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, and a Fellow in the Computer Science Programme of the Singapore-MIT Alliance. He received his BSc. in computer science from Brown University in 1984, and his Ph.D in computer science from Stanford University in 1994. His research interests and achievements can be found at his homepage ⟨http://www.cag.lcs.mit.edu/~rinard/⟩.