Typed Regions for Tag-Free Garbage Collection

Martin Elsman*

IT University of Copenhagen.
Glentevej 67, DK-2400 Copenhagen NV, Denmark.
Email: mael@itu.dk

October, 2002

Abstract. This paper presents an extension to the Tofte-Talpin region type system, which makes it possible to combine region-based memory management with a partly tag-free garbage collection algorithm. The memory discipline is implemented for Standard ML in the ML Kit compiler. Experiments show that, for a range of benchmark programs, the discipline improves performance, both with respect to memory usage and execution time.

1 Introduction

In implementations of languages that support dynamic reference tracing garbage collection, such as most Java and ML implementations, tags are often used to distinguish between different types of values at runtime. The use of tags imposes a memory overhead, however, which for the most commonly used data structures, such as tuples, lists, and trees, can be as large as one third of the total memory usage.

In this paper we present a memory discipline that integrates region-based memory management and reference tracing garbage collection in such a way that tagging of commonly used values such as tuples and lists is avoided.

In the Tofte-Talpin region typing rules [18], two values are forced into the same region, only if their types are identical. This property suggests that, in systems based on the Tofte-Talpin region typing rules, tags to distinguish different types of values at runtime can be moved from individual values to the level of regions. We show that this "Big Bag of Pages" approach to avoid tagging improves memory usage and execution time for many programs.

1.1 Combining Region Inference and Garbage Collection

In the region-based memory model, the store is organized as a stack of dynamically expandable regions. At compile time, allocation and deallocation directives are added to the program by an analysis called region inference. Each value-creating expression is annotated with information that directs in what region

* Part time at Royal Veterinary and Agricultural University, Denmark

the value goes at runtime. Moreover, if e is some region-annotated expression, then so is `letregion` ρ `in` e `end`. An expression of this form is evaluated by first allocating a region on top of the region stack and binding the region to the region variable ρ. Then e is evaluated, possibly using the region bound to ρ for holding values. Finally, upon reaching `end`, the region is reclaimed.

In previous work, Hallenberg et al. [9] integrated garbage collection with region inference and showed that, for most programs not optimized for region inference, adding garbage collection reduces memory usage significantly. Moreover, from the point of view of garbage collection, region inference reduces the pressure on the garbage collector.

The tag-based garbage collection algorithm described in [9] is a generalization of Cheney's copying garbage collection algorithm. Each region is collected independently: if a value survives a garbage collection, the value belongs to the same region as before the garbage collection.

1.2 Tag-free Garbage Collection

There are a series of proposals for tag-free garbage collection schemes [2, 7, 8, 19, 1] and nearly tag-free garbage collection schemes [14, 13].

The *partly tag-free garbage collection* scheme that we present here does not involve untagging of all values. In particular, unboxed objects (e.g., integers and booleans) are tagged in our system, which makes it possible to distinguish pointers from unboxed objects at runtime. However, the scheme allows for commonly used data structures, such as tuples, reals, and reference cells, to be untagged, which can lead to significant time and memory savings. In this work, we investigate the effect of untagging pairs, which are used for the implementation of many dynamic data structures, including lists.

As is the case for other techniques that support full untagging, our technique does not involve traversing the runtime stack to determine types during a garbage collection [2, 7, 8] or require special type information to be passed to functions at runtime [19]. By requiring values in certain regions to be of the same kind, our approach has much in common with BIBOP (Big Bag Of Pages) systems, with regions as pages [10].

1.3 Contributions of this Paper

The contributions of this paper are threefold. First, in Sect. 2, we present a region type system that guarantees that regions containing pairs contain no values of other kinds. Second, based on Cheney's algorithm for regions [9], which is summarized in Sect. 3, we present a partly tag-free garbage collection algorithm in Sect. 4. Third, in Sect. 5, we provide evidence that the algorithm improves execution times and memory usage compared to the tag-based algorithm. The paper ends with a conclusion and directions for future work.

2 A Type System with Typed Regions

In this section, we present a type system that forces values of different kinds of types into different regions. There are two essential ways that our type system differs from the Tofte-Talpin type system [18]. First, the Tofte-Talpin type system does not restrict two values from residing in the same region. Second, because our system uses a combination of reference tracing garbage collection and region based memory management, the type system is extended to ensure that no dangling pointers are introduced during evaluation [6].

2.1 Region Types, Variables, and Effects

A *region type*, ranged over by κ, is either the token `pair`, which classifies regions containing pairs, or the token `other`, which classifies regions containing other values than pairs and integers. Integers are unboxed and thus do not reside in distinguished regions.

We assume a denumerably infinite set of *region variables*, ranged over by ρ. Each region variable has associated with it a region type. We write $rt(\rho)$ to refer to the region type associated with ρ. We also assume a denumerably infinite set of *effect variables*, ranged over by ϵ, a denumerably infinite set of *type variables*, ranged over by α, and a denumerably infinite set of *program variables*, ranged over by x and f. An *atomic effect*, ranged over by η, is either a region variable or an effect variable. An *arrow effect*, written $\epsilon.\varphi$, is a pair of an effect variable and a set φ of atomic effects.

2.2 Types and Substitutions

The grammars for *types* (τ), *type and places* (μ), *type schemes* (σ), and *type scheme and places* (π) are as follows:

$$\begin{array}{lll} \mu & ::= & (\tau,\rho) \mid \alpha \mid \texttt{int} \\ \sigma & ::= & \forall\vec{\alpha}\vec{\epsilon}\vec{\rho}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2 \end{array} \qquad \begin{array}{lll} \tau & ::= & \mu_1 \times \mu_2 \mid \mu_1 \xrightarrow{\epsilon.\varphi} \mu_2 \\ \pi & ::= & (\sigma,\rho) \mid \mu \end{array}$$

A type scheme and place (or type and place) π is *well-formed* if the sentence $\vdash \pi$ can be derived from the following rules:

$$\vdash \alpha \qquad \vdash \texttt{int} \qquad \frac{\mathrm{rt}(\rho) = \texttt{pair}}{\vdash (\mu_1 \times \mu_2, \rho)} \qquad \frac{\mathrm{rt}(\rho) = \texttt{other}}{\vdash (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho)} \qquad \frac{\vdash (\tau,\rho)}{\vdash (\forall\vec{\rho}\vec{\alpha}\vec{\epsilon}.\tau,\rho)}$$

A *region substitution* (S^{r}) is a finite map from region variables to region variables, such that $\mathrm{rt}(\rho) = \mathrm{rt}(S^{\mathrm{r}}(\rho))$ for any region variable $\rho \in \mathrm{dom}(S^{\mathrm{r}})$. A *substitution* ($S$) is a triple $(S^{\mathrm{r}}, S^{\mathrm{t}}, S^{\mathrm{e}})$, where S^{r} is a region substitution, S^{t} is a finite map from type variables to well-formed type and places, and S^{e} is a finite map from effect variables to arrow effects. The effect of applying a substitution on a particular object is to carry out the three substitutions simultaneously on the three kinds of variables in the object (possibly by renaming of bound variables within

the object to avoid capture). For effect sets and arrow effects, substitution is defined as follows [15], assuming $S = (S^{\mathrm{r}}, S^{\mathrm{t}}, S^{\mathrm{e}})$:

$$S(\varphi) = \{S^{\mathrm{r}}(\rho) \mid \rho \in \varphi\} \cup \{\eta \mid \exists \epsilon, \epsilon', \varphi'.\epsilon \in \varphi \wedge S^{\mathrm{e}}(\epsilon) = \epsilon'.\varphi' \wedge \eta \in \{\epsilon'\} \cup \varphi'\}$$

$$S(\epsilon.\varphi) = \epsilon'.(\varphi' \cup S(\varphi)), \text{ where } S^{\mathrm{e}}(\epsilon) = \epsilon'.\varphi'$$

One can show that well-formedness is closed under substitution; if $\vdash \pi$ then $\vdash S(\pi)$, for any substitution S.

A type scheme $\sigma = \forall\vec{\rho}\vec{\alpha}\vec{\epsilon}.\tau'$ *generalizes* a type τ *via* $\vec{\rho}'$, written $\sigma \geq \tau$ via $\vec{\rho}'$, if there exists a substitution $S = (\{\vec{\rho}'/\vec{\rho}\}, S^{\mathrm{t}}, S^{\mathrm{e}})$ such that $S(\tau') = \tau$ and $\mathrm{dom}(S^{\mathrm{t}}) = \{\vec{\alpha}\}$, and $\mathrm{dom}(S^{\mathrm{e}}) = \{\vec{\epsilon}\}$. If $\sigma \geq \tau$ via $\vec{\rho}$, for some σ, τ, and $\vec{\rho}$, and S is a substitution, then $S(\sigma) \geq S(\tau)$ via $S(\vec{\rho})$.

A *type environment* (Γ) is a finite map from program variables to type scheme and places. Following the usual definition of bound variables, we define, for any kind of object o, the *free region variables* and the *free region and effect variables* of o, written $\mathrm{frv}(o)$ and $\mathrm{frev}(o)$, respectively. We write $\mathrm{fv}(o)$ to denote the *free type, region, and effect variables* of o.

2.3 Terms

The grammars for *expressions* (e) and *values* (v) are as follows:

$$\begin{array}{rcl} v & \texttt{::=} & d \mid (v_1, v_2)\ \texttt{in}\ \rho \mid \lambda x.e\ \texttt{in}\ \rho \mid \texttt{fun}\ f\ \texttt{[}\vec{\rho}\texttt{]}\ x\ \texttt{=}\ e\ \texttt{in}\ \rho \\ e & \texttt{::=} & v \mid x \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mid e_1\ e_2 \mid \lambda x.e\ \texttt{at}\ \rho \mid \texttt{letregion}\ \rho\ \texttt{in}\ e \\ & \mid & \texttt{fun}\ f\ \texttt{[}\vec{\rho}\texttt{]}\ x\ \texttt{=}\ e\ \texttt{at}\ \rho \mid e\ \texttt{[}\vec{\rho}\texttt{]}\ \texttt{at}\ \rho \mid (e_1, e_2)\ \texttt{at}\ \rho \mid \texttt{\#}i\ e \end{array}$$

All values, except integers, are boxed and associated with distinguished regions. The *free (program) variables* of some expression (or value) e is written $\mathrm{fpv}(e)$.

2.4 Typing Rules

To guarantee safety of garbage collection, we must ensure that no dangling pointers are introduced during evaluation, which is not guaranteed by the Tofte-Talpin region type system [18]. The solution that we apply here is to add additional side conditions to the typing rules for functions that guarantee the absence of dangling pointers [6].

First, we define a notion of *value containment*; all values in an expression e is contained in a set of regions φ with appropriate region types, if the sentence $\varphi \models_{\mathrm{v}} e$ is derivable from the rules in Fig. 1.

We now introduce a relation $\mathcal{G}$, which we shall use to strengthen the typing rules for functions to avoid dangling pointers during evaluation. The relation is derived from the side condition for functions suggested by Tofte and Talpin in [17, page 50]. The relation $\mathcal{G}$ is parameterized over an environment Γ, a function body e, a set of function parameters X, and the type scheme and place π of the function:

$$\mathcal{G}(\Gamma, e, X, \pi) \;=\; \forall y \in \mathrm{fpv}(e) \setminus X.\mathrm{frev}(\Gamma(y)) \subseteq \mathrm{frev}(\pi) \;\wedge\; \mathrm{frv}(\pi) \models_{\mathrm{v}} e$$

$$\varphi \models_{\mathrm{v}} d \quad \frac{\varphi \models_{\mathrm{v}} e \quad \rho \in \varphi \quad \mathrm{rt}(\rho) = \texttt{other}}{\varphi \models_{\mathrm{v}} \lambda x.e \texttt{ in } \rho} \quad \frac{\varphi \models_{\mathrm{v}} v_1 \quad \varphi \models_{\mathrm{v}} v_2 \quad \rho \in \varphi \quad \mathrm{rt}(\rho) = \texttt{pair}}{\varphi \models_{\mathrm{v}} (v_1, v_2) \texttt{ in } \rho} \quad \frac{\rho \in \varphi \quad \varphi \models_{\mathrm{v}} e \quad \mathrm{rt}(\rho) = \texttt{other}}{\varphi \models_{\mathrm{v}} \texttt{fun } f\ [\vec{\rho}]\ x = e \texttt{ in } \rho}$$

$$\varphi \models_{\mathrm{v}} x \quad \frac{\varphi \models_{\mathrm{v}} e_1 \quad \varphi \models_{\mathrm{v}} e_2}{\varphi \models_{\mathrm{v}} (e_1, e_2) \texttt{ at } \rho} \quad \frac{\varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{v}} \lambda x.e \texttt{ at } \rho} \quad \frac{\varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{v}} \#i\ e}$$

$$\frac{\varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{v}} \texttt{fun } f\ [\vec{\rho}]\ x = e \texttt{ at } \rho} \quad \frac{\varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{v}} e\ [\vec{\rho}] \texttt{ at } \rho} \quad \frac{\varphi \models_{\mathrm{v}} e_1 \quad \varphi \models_{\mathrm{v}} e_2}{\varphi \models_{\mathrm{v}} e_1\ e_2}$$

$$\frac{\varphi \models_{\mathrm{v}} e_1 \quad \varphi \models_{\mathrm{v}} e_2}{\varphi \models_{\mathrm{v}} \texttt{let } x = e_1 \texttt{ in } e_2} \quad \frac{\rho \notin \varphi \quad \varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{v}} \texttt{letregion } \rho \texttt{ in } e}$$

Fig. 1. Value containment.

The typing rules are shown in Fig. 2 and allow inference of sentences of the form $\Gamma \vdash e : \pi, \varphi$, which are read: in the type environment Γ, the expression (or value) e has type scheme and place π and effect φ.

The typing rules are closed under substitution; if $\Gamma \vdash e : \pi, \varphi$ then $S(\Gamma) \vdash S(e) : S(\pi), S(\varphi)$, for any substitution S. This property relies on the garbage collection safety relation being closed under substitution.

2.5 A Small Step Dynamic Semantics

The dynamic semantics that we present is in the style of a contextual dynamic semantics [12] and is similar to the semantics given by Helsen and Thiemann [11, 4], although it differs in the way that inaccessibility to values in deallocated regions are modeled. Whereas Helsen and Thiemann "null out" references to deallocated regions (to avoid future access), our semantics keep track of a set of currently allocated regions and disallow access to regions that are not in this set.

The grammars for *evaluation contexts* (E) and *instructions* (ι) are as follows:

$$\begin{array}{lll}
E_\varphi & ::= & [\cdot] \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\varphi = \emptyset) \\
& | & \texttt{letregion } \rho \texttt{ in } E_{\varphi\setminus\{\rho\}} \qquad\qquad\qquad\qquad (\rho \in \varphi) \\
& | & E_\varphi\ e \ \mid\ v\ E_\varphi \ \mid\ E_\varphi\ [\vec{\rho}] \texttt{ at } \rho \ \mid\ \texttt{let } x = E_\varphi \texttt{ in } e \\
& | & (E_\varphi, e) \texttt{ at } \rho \ \mid\ (v, E_\varphi) \texttt{ at } \rho \ \mid\ \#i\ E_\varphi \\
\iota & ::= & d \ \mid\ \lambda x.e \texttt{ at } \rho \ \mid\ (v_1, v_2) \texttt{ at } \rho \\
& | & \#1\ ((v_1, v_2) \texttt{ in } \rho) \ \mid\ \#2\ ((v_1, v_2) \texttt{ in } \rho) \\
& | & (\lambda x.e \texttt{ in } \rho)\ v \ \mid\ (\texttt{fun } f\ [\vec{\rho}]\ x = e \texttt{ in } \rho)\ [\vec{\rho}'] \texttt{ at } \rho
\end{array}$$

Contexts E_φ make explicit the set of regions φ bound by `letregion` constructs that encapsulate the hole in the context.

The evaluation rules are given in Fig. 3 and consist of *allocation and deallocation rules*, *reduction rules*, and a *context rule*. The rules are of the form $e \overset{\varphi}{\longmapsto} e'$,

Values $\boxed{\Gamma \vdash v : \pi, \varphi}$

$$\frac{}{\Gamma \vdash d : \texttt{int}, \emptyset} \qquad \frac{\begin{array}{c}\{x : \mu_1\} \vdash e : \mu_2, \varphi \\ \vdash \mu \quad \mu = (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \\ \mathcal{G}(\Gamma, e, \{x\}, \mu)\end{array}}{\Gamma \vdash \lambda x.e \texttt{ in } \rho : \mu, \emptyset} \qquad \frac{\begin{array}{c}rt(\rho) = \texttt{pair} \\ \Gamma \vdash v_1 : \mu_1, \emptyset \quad \Gamma \vdash v_2 : \mu_2, \emptyset\end{array}}{\Gamma \vdash (v_1, v_2) \texttt{ in } \rho : (\mu_1 \times \mu_2, \rho), \emptyset}$$

$$\frac{\begin{array}{c}\{f : (\forall \vec{\rho}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), x : \mu_1\} \vdash e : \mu_2, \varphi \quad \vdash \pi \\ \mathrm{fv}(\vec{\alpha}\vec{\epsilon}\vec{\rho}) \cap \mathrm{fv}(\Gamma, \varphi) = \emptyset \quad \pi = (\forall \vec{\alpha}\vec{\epsilon}\vec{\rho}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \quad \mathcal{G}(\Gamma, e, \{f, x\}, \pi)\end{array}}{\Gamma \vdash \texttt{fun } f \texttt{ [}\vec{\rho}\texttt{]} \ x \texttt{ = } e \texttt{ in } \rho : \pi, \emptyset}$$

Expressions $\boxed{\Gamma \vdash e : \pi, \varphi}$

$$\frac{\Gamma \vdash e : \pi, \varphi \quad \varphi' \supseteq \varphi}{\Gamma \vdash e : \pi, \varphi'} \qquad \frac{\begin{array}{c}\Gamma + \{x : \mu_1\} \vdash e : \mu_2, \varphi \quad \vdash \mu \\ \mu = (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \quad \mathcal{G}(\Gamma, e, \{x\}, \mu)\end{array}}{\Gamma \vdash \lambda x.e \texttt{ at } \rho : \mu, \{\rho\}} \qquad \frac{\Gamma(x) = \pi \quad \vdash \pi}{\Gamma \vdash x : \pi, \emptyset}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : (\sigma, \rho'), \varphi \\ \sigma \geq \tau \text{ via } \vec{\rho} \quad \vdash (\tau, \rho)\end{array}}{\Gamma \vdash e \texttt{ [}\vec{\rho}\texttt{]} \texttt{ at } \rho : (\tau, \rho), \varphi \cup \{\rho, \rho'\}} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, \rho), \varphi_1 \\ \Gamma \vdash e_2 : \mu', \varphi_2\end{array}}{\Gamma \vdash e_1 \ e_2 : \mu, \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}}$$

$$\frac{\begin{array}{c}rt(\rho) = \texttt{pair} \\ \Gamma \vdash e_1 : \mu_1, \varphi_1 \quad \Gamma \vdash e_2 : \mu_2, \varphi_2\end{array}}{\Gamma \vdash (e_1, e_2) \texttt{ at } \rho : (\mu_1 \times \mu_2, \rho), \varphi_1 \cup \varphi_2 \cup \{\rho\}} \qquad \frac{\Gamma \vdash e : (\mu_1 \times \mu_2, \rho), \varphi \quad i \in \{1, 2\}}{\Gamma \vdash \texttt{\#}i \ e : \mu_i, \varphi \cup \{\rho\}}$$

$$\frac{\Gamma \vdash e : \mu, \varphi \quad \rho \notin \mathrm{frev}(\Gamma, \mu)}{\Gamma \vdash \texttt{letregion } \rho \texttt{ in } e : \mu, \varphi \setminus \{\rho\}} \qquad \frac{\Gamma \vdash e_1 : \pi, \varphi_1 \quad \Gamma + \{x : \pi\} \vdash e_2 : \mu, \varphi_2}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \mu, \varphi_1 \cup \varphi_2}$$

$$\frac{\begin{array}{c}\Gamma + \{f : (\forall \vec{\rho}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), x : \mu_1\} \vdash e : \mu_2, \varphi \quad \vdash \pi \\ \mathrm{fv}(\vec{\alpha}\vec{\epsilon}\vec{\rho}) \cap \mathrm{fv}(\Gamma, \varphi) = \emptyset \quad \pi = (\forall \vec{\alpha}\vec{\epsilon}\vec{\rho}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \quad \mathcal{G}(\Gamma, e, \{f, x\}, \pi)\end{array}}{\Gamma \vdash \texttt{fun } f \texttt{ [}\vec{\rho}\texttt{]} \ x \texttt{ = } e \texttt{ at } \rho : \pi, \{\rho\}}$$

Fig. 2. Typing rules.

which says that, given a set of allocated regions φ, the expression e reduces (in one step) to the expression e'. Next, the *evaluation* relation $\overset{\varphi}{\longmapsto}{}^*$ is defined as the least relation formed by the reflexive transitive closure of the relation $\overset{\varphi}{\longmapsto}$. We further define $e \Downarrow_\varphi v$ to mean $e \overset{\varphi}{\longmapsto}{}^* v$, and $e \Uparrow_\varphi$ to mean that there exists an infinite sequence, $e \overset{\varphi}{\longmapsto} e_1 \overset{\varphi}{\longmapsto} e_2 \overset{\varphi}{\longmapsto} \cdots$.

2.6 Type Safety

The proof of type safety is based on well-known techniques for proving type safety for statically typed languages [12, 20]. We shall not present the complete proofs here, but refer the reader to [6], which includes proofs for a similar system.

Allocation and Deallocation $\boxed{e \overset{\varphi}{\longmapsto} v}$

$$\lambda x.e \texttt{ at } \rho \overset{\varphi \cup \{\rho\}}{\longmapsto} \lambda x.e \texttt{ in } \rho \qquad (v_1, v_2) \texttt{ at } \rho \overset{\varphi \cup \{\rho\}}{\longmapsto} (v_1, v_2) \texttt{ in } \rho$$

$$\texttt{fun } f\ [\vec{\rho}]\ x = e \texttt{ at } \rho \overset{\varphi \cup \{\rho\}}{\longmapsto} \texttt{fun } f\ [\vec{\rho}]\ x = e \texttt{ in } \rho \qquad \texttt{letregion } \rho \texttt{ in } v \overset{\varphi}{\longmapsto} v$$

Reduction and Context $\boxed{e \overset{\varphi}{\longmapsto} e'}$

$$(\lambda x.e \texttt{ in } \rho)\ v \overset{\varphi \cup \{\rho\}}{\longmapsto} e[v/x] \qquad \texttt{let } x = v \texttt{ in } e \overset{\varphi}{\longmapsto} e[v/x]$$

$$(\texttt{fun } f\ [\vec{\rho}]\ x = e \texttt{ in } \rho)[\vec{\rho}'] \texttt{ at } \rho' \overset{\varphi \cup \{\rho\}}{\longmapsto} \lambda x.e[\vec{\rho}'/\vec{\rho}][(\texttt{fun } f\ [\vec{\rho}]\ x = e \texttt{ in } \rho)/f] \texttt{ at } \rho'$$

$$\#1\ ((v_1, v_2) \texttt{ in } \rho) \overset{\varphi \cup \{\rho\}}{\longmapsto} v_1 \qquad \#2\ ((v_1, v_2) \texttt{ in } \rho) \overset{\varphi \cup \{\rho\}}{\longmapsto} v_2$$

$$\frac{e \overset{\varphi' \cup \varphi}{\longmapsto} e' \quad \varphi \cap \varphi' = \emptyset \quad E_\varphi \neq [\cdot]}{E_\varphi[e] \overset{\varphi'}{\longmapsto} E_\varphi[e']}\ (1)$$

Fig. 3. Evaluation rules.

We first state a property saying that a well-typed expression is either a value or can be separated into an evaluation context and an instruction:

Proposition 1 (Unique Decomposition). *If $\vdash e : \pi, \varphi$, then either (1) e is a value, or (2) there exists a unique $E_{\varphi'}$, e', and π' such that $e = E_{\varphi'}[e']$ and $\vdash e' : \pi', \varphi \cup \varphi'$ and e' is an instruction.*

Proof. By induction on the structure of e. □

A type preservation property (i.e., subject reduction) for the language can be stated as follows:

Proposition 2 (Preservation). *If $\vdash e : \pi, \varphi$ and $e \overset{\varphi}{\longmapsto} e'$ then $\vdash e' : \pi, \varphi$.*

Proof. By induction on the derivation $e \overset{\varphi}{\longmapsto} e'$. □

Proposition 3 (Progress). *If $\vdash e : \pi, \varphi$ then either e is a value or else there exists some e' such that $e \overset{\varphi}{\longmapsto} e'$.*

Proof. If e is not a value, then by Proposition 1 there exists a unique $E_{\varphi'}$, ι, and π' such that $e = E_{\varphi'}[\iota]$ and $\vdash \iota : \pi', \varphi \cup \varphi'$. The remainder of the proof argues that $\iota \overset{\varphi \cup \varphi'}{\longmapsto} e''$, for some e'', so that $E_{\varphi'}[\iota] \overset{\varphi}{\longmapsto} E_{\varphi'}[e'']$ follows from (1). □

Theorem 4 (Type Soundness). *If $\vdash e : \pi, \varphi$, then either $e \Uparrow_\varphi$ or else there exists some v such that $e \Downarrow_\varphi v$ and $\vdash v : \pi, \varphi$.*

Proof. By induction on the number of rewriting steps, using Proposition 2 and Proposition 3. □

We now introduce the notion of *context containment*, written $\varphi \models_{\mathrm{c}} e$, which expresses that when e can be written on the form $E_{\varphi'}[e']$, values in e' may be contained in regions in the set $\varphi \cup \varphi'$, where φ' are regions on the stack represented by the evaluation context $E_{\varphi'}$. The definition of context containment is given in Fig. 4. The following containment theorem states that, for well-typed programs, containment is preserved under evaluation:

$$\varphi \models_{\mathrm{c}} x \qquad \frac{\varphi \models_{\mathrm{v}} v}{\varphi \models_{\mathrm{c}} v} \qquad \frac{\rho \notin \varphi \quad \varphi \cup \{\rho\} \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} \texttt{letregion}\ \rho\ \texttt{in}\ e} \qquad \frac{\varphi \models_{\mathrm{c}} e \quad \varphi \models_{\mathrm{v}} e'}{\varphi \models_{\mathrm{c}} \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'}$$

$$\frac{\varphi \models_{\mathrm{c}} e \quad \varphi \models_{\mathrm{v}} e'}{\varphi \models_{\mathrm{c}} e\ e'} \qquad \frac{\varphi \models_{\mathrm{v}} v \quad \varphi \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} v\ e} \qquad \frac{\varphi \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} e\ \texttt{[}\vec{\rho}\texttt{]}\ \texttt{at}\ \rho}$$

$$\frac{\varphi \models_{\mathrm{c}} e \quad \varphi \models_{\mathrm{v}} e'}{\varphi \models_{\mathrm{c}} \texttt{(}e, e'\texttt{)}\ \texttt{at}\ \rho} \qquad \frac{\varphi \models_{\mathrm{v}} v \quad \varphi \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} \texttt{(}v, e\texttt{)}\ \texttt{at}\ \rho} \qquad \frac{\varphi \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} \texttt{\#}i\ e}$$

Fig. 4. Context containment.

Theorem 5 (Containment). *If* $\vdash e : \pi, \varphi$ *and* $\varphi' \models_{\mathrm{c}} e$ *and* $e \overset{\varphi'}{\longmapsto} e'$ *then* $\varphi' \models_{\mathrm{c}} e'$.

Proof. By induction on the structure of e. □

The containment theorem states that evaluation allocates only in regions that are either global or present on the region stack, represented by the evaluation context. Moreover, at any point during evaluation, no region contains a value that does not conform to the region type of the region.

3 Garbage Collecting Regions

Before we present the partly tag-free garbage collection algorithm, we first give an overview of the tag-based garbage collection algorithm [9], on which the partly tag-free algorithm is built.

The store consists of a *stack* and, separate from the stack, a *region heap*. The stack consists of *activation records*, which are pushed on the stack at function entry and popped on exit. The sizes of activation records are determined statically. The region heap consists of a set of fixed-size *region pages*, some of which are linked together in a *free-list*. At runtime we distinguish between two kinds of regions [3]:

Finite regions. Regions inferred to hold only one value at a time. The size of the region is the maximal size of the values that may be allocated in the region. Finite regions are allocated in activation records on the stack and usually contain tuples and closures.

Infinite regions. Regions inferred to hold an unbounded number of values. An infinite region is represented by a linked list of region pages, pointed to by a *region descriptor*, which resides in an activation record on the stack. Infinite regions usually contain lists and other recursive data structures.

Values that fit in one word, such as integers, are implemented unboxed and therefore do not reside in distinguished regions.

3.1 Region Pages

Every region page starts with a *region page descriptor*. It contains a pointer to the next region page in the region. It also contains an *origin pointer*, which points back to the region descriptor of the region.

All region pages used by regions have the same fixed size (1 kb) and are aligned at 1 kb boundaries, which makes it possible to determine, given a pointer to some value, the region page header for the region page holding the value. The function `regiondesc`(p) returns the region descriptor associated with the region holding the value pointed to by p by accessing the associated region page descriptor as described above and then extracting the origin pointer from it.

3.2 Infinite Regions and Region Descriptors

A region descriptor is a quadruple (e, fp, a, rs), where e is an *end pointer* pointing to the end of the most recently allocated page in the region; a is an *allocation pointer* pointing to the first available free location in the most recently allocated page in the region; fp is the *first-page pointer* pointing to the first page of the region; and rs is a *region status*, which is a two-valued mark for identifying, during a garbage collection, if there are values to scan in the region. In the implementation, the region status mark occupies only one bit and is encoded in one of the pointer fields in the region descriptor. Figure 5 shows an example runtime stack containing three region descriptors and one finite region.

Allocating a value is done at a if there is enough space in the region page; otherwise, the region is extended with a region page taken from the free-list. An infinite region is allocated by requesting a region page from the free-list and updating a region descriptor. When an infinite region is popped, its region pages are appended to the free-list; this operation can be done in constant time.

3.3 Cheney's Algorithm for Regions

The tag-based garbage collection algorithm is based on Cheney's stop and copy algorithm [5]. Intuitively, each region is associated with a from-space and a to-space. When (e, fp, a, rs) is some region descriptor, a plays the dual role of the

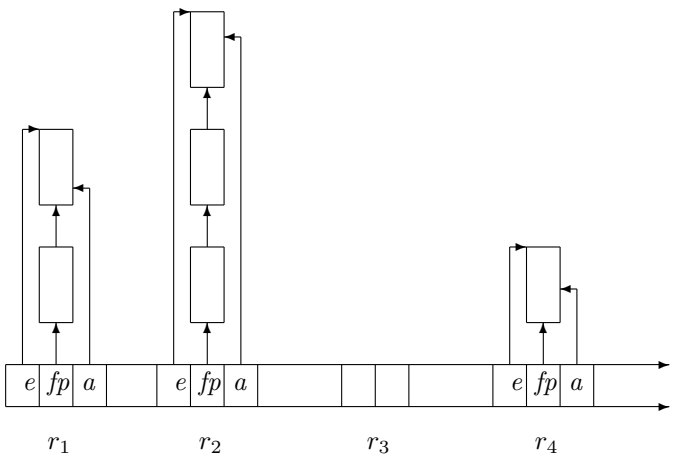

Fig. 5. A runtime stack containing three region descriptors and a finite region (r_3).

allocation pointer for the mutator and the allocation pointer for the garbage collector. Scan pointers are kept in a scan stack; there is no scan pointer in the region descriptor. In the following, when r is some region descriptor, we use the notation $r \to a$ to refer to the allocation pointer in r; we use similar notation to access the other components of a region descriptor.

Cheney's algorithm for regions applies Cheney's algorithm locally on each region and uses the stop criteria: $\forall r \in \mathit{Reg} : (r \to a) = s_r$, where Reg is the set of region descriptors on the stack and s_r is the scan pointer of r. The stop criteria is implemented using the *scan stack*, which consists of those scan pointers s_r for which $s_r \neq (r \to a)$.

The garbage collector never allocates into from-spaces. At the start of a garbage collection, the region stack is traversed and region pages in from-space areas (pointed at by $r \to \mathit{fp}$) are linked together to form a single global from-space area. Next, for every region descriptor r on the stack, $r \to \mathit{fp}$ is initialized to point at a fresh region page taken from the free-list. Moreover, $r \to a$ is initialized to point at the beginning of the page pointed to by $r \to \mathit{fp}$ and $r \to e$ at the end of the page pointed to by $r \to \mathit{fp}$. While collection is in progress, region pages are allocated from the free-list, which is disjoint from the global from-space area. After garbage collection, the global from-space area is appended to the free-list in a constant-time operation.

The Cheney algorithm extended to regions is outlined in Fig. 6. In a call `cheney`(r, s, a), the argument r is the address of a region descriptor of a region; it plays the role of the to-space. The `for`-loop scans the value pointed at by s and calls the function `evacuate` on all fields inside the value. The call `next_value`(s, r) proceeds to the value following the value pointed at by s in r (which may entail proceeding to the next region page in the region).

```
fun evacuate(p) {
  if ( is_int(p) ) return p;
  if ( is_fwd_ptr(*p) )
    return *p;
  r = regiondesc(p);
  a = acopy(r,p);
  if ( r->rs == NONE ) {
     r->rs = SOME;
     push_onto_scanstack(a);
  }
  *p = a; // set fwd-ptr
  return a;
}
```

```
fun cheney(r,s,a) {
  while ( s != a ) {
    for ( i=1; i<sz_tag(*s); i++ )
      *(s+i) = evacuate(*(s+i));
    s = next_value(s,r);
  }
  r->rs = NONE;
}
fun collect_regions() {
  while ( s = pop_scanstack() ) {
    r = regiondesc(s);
    cheney(r, s, r->a);
  }
```

Fig. 6. Cheney's algorithm for regions.

Next consider the function `evacuate`. Integers passed to the function are returned unmodified. Given a pointer p to a value in from space, the function determines if a forward pointer is installed, in which case the forward pointer is returned. To determine if a forward pointer is installed, the function `is_fwd_ptr` is used, which examines the tag word identified by the pointer p to see if it is a pointer (i.e., that the two lower bits are zero).

If no forward pointer is installed, the function `regiondesc`(p) returns the address of the region descriptor of the region containing the value pointed to by p (see Sect. 3.1). The function `acopy`(r, p) allocates a copy of the value pointed to by p in the region described by r and returns the address of the new copy. (`acopy` extends the region with a new page, if necessary.) The region status field rs in the region descriptor is `NONE` if $s_r = (r \to a)$ and `SOME` if $s_r \neq (r \to a)$. The field rs is set to `NONE` in function `cheney` when the queue of unscanned values in the region becomes empty. Notice that because a region is composed of region pages, it is not always the case that $s_r \leq (r \to a)$, but $s_r = (r \to a)$ does signify that all values in the region have been scanned.

The algorithm maintains the invariant that the region status $r \to rs$ of some region descriptor r is `SOME` if and only if either s_r is on the scan stack or r denotes a region that is currently being scanned. In the implementation, the region status is encoded as a bit in one of the pointer fields in the region descriptor.

Finally, `collect_regions` repeatedly calls `cheney` on one region at a time until the scan stack is empty.

4 Tag-free Collection of Pairs

Based on the tag-based garbage collection algorithm outlined in the previous section, we now present a refined algorithm for partly tag-free garbage collection of regions. We first refine the layout of region page descriptors and region descriptors as follows:

– Region page descriptors are refined to include a *space status*, which specifies whether the page is part of to-space or part of from-space. Using the alignment properties and the space status of region pages, the function `in_tospace`(p) can determine if an object pointed to by p is located in a region page in to-space or from-space.
– Region descriptors are refined to include a region type, which allows the function `is_pairregion`(r) to determine if a region contains untagged pairs.

The space status of a region page descriptor and the region type of a region descriptor can be encoded as bits in pointer fields in the respective descriptors.

When pairs are untagged, a pair value is represented by a pointer to a word in memory preceding the two words that make up the pair. Thus, from the mutator's point of view, the pair looks as if it is tagged. However, the mutator never tries to inspect the tag—it can use only the components of the pair. The garbage collection algorithm, however, can use the `is_pairregion` function to determine if the value is a pair and then arrange that forward pointers into to-space are stored in one of the components of the pair. Following these ideas, the `evacuate` and `cheney` functions are refined as shown in Fig. 7.

```
fun evacuate(p) {
  if ( is_int(p) ) return p;
  r = regiondesc(p);
  if ( is_pairregion(r) ) {
    if ( in_tospace(*(p+1)) )
      return *(p+1);
    a = acopy_pair(r,p);
    *(p+1) = a; // set fwd-ptr
  } else {
    if ( is_fwd_ptr(*p) )
      return *p;
    a = acopy(r,p);
    *p = a;     // set fwd-ptr
  }
  if ( r->rs == NONE ) {
    r->rs = SOME;
    push_onto_scanstack(a);
  }
  return a;
}
```

```
fun cheney(r,s,a) {
  if ( is_pairregion(r) ) {
    while ( s+1 != a ) {
      *(s+1) = evacuate(*(s+1));
      *(s+2) = evacuate(*(s+2));
      s = next_pair(s,r);
    }
  } else {
    while ( s != a ) {
      for ( i=1; i<sz_tag(*s); i++ )
        *(s+i) = evacuate(*(s+i));
      s = next_value(s,r);
    }
  }
  r->rs = NONE;
}
```

Fig. 7. Tag-free collection of pairs.

Two new auxiliary functions `next_pair` and `acopy_pair` are used, which are identical to the functions `next_value` and `acopy`, although specialized for pairs.

4.1 Implementation Details

The partly tag-free garbage collection algorithm is implemented for Standard ML in the ML Kit compiler [16]. Besides from modifying the code generator not to emit code for storing tags in infinite regions of type `pair`, the algorithm is extended to handle finite regions and so-called *large objects*, which are objects that do not fit in region pages.

Although finite regions are not collected by the garbage collector, the garbage collector does scan values in finite regions, which requires tagging of such values. From the point of view of the mutator, there is no difference between pairs in finite regions and pairs in infinite regions, except that the mutator writes a tag word when it stores a pair into a finite region. The details of extending the algorithm to work for finite regions, without traversing values in finite regions more than once, are outlined in [9].

To manage large objects efficiently and to allow efficient natural representations of certain data types, such as strings and arrays, large objects associated with a region are allocated using `malloc` and stored in a linked list, pointed to from a field in the region descriptor. Upon deallocation of a region, large objects in the associated linked list are deallocated using `free`. Large objects are never copied by the garbage collector but may need to be scanned by the collector. To determine whether a pointer points to a large object or an object in a region, large objects are tagged and aligned at 1 kb boundaries to distinguish them from region pages.

5 Measurements

In this section, we present evidence that our tag-free garbage collection algorithm improves execution times and memory usage compared to the fully tag-based garbage collection algorithm for regions [9].

All benchmark programs are run on a 750Mhz Pentium III Linux box with 512Mb RAM. Times reported are user CPU times and memory usage is the sum of the maximum stack size, the maximum size of allocated large objects, and the maximum size of allocated region pages, as reported by the runtime system. Experiments are performed with the ML Kit version 4.1.2 [16] for a range of benchmark programs, spanning from small micro-benchmarks (`msort`) to larger programs, such as `vliw` and `mlyacc`.

The benchmark results are shown in Fig. 8. The first column shows the size of each benchmark in lines of source code. The next three columns show the memory usage with and without tagging of values and the improvements in percentages. The following three columns (**# GC**) show the number of garbage collection for each of the two settings and the improvements in percentages. With one exception (i.e., `tsp`), untagging of pairs has a positive effect on either memory usage or the number of garbage collections (the `tsp` benchmark makes no use of pairs.) The `mlyacc` benchmark uses slightly less memory when pairs are tagged than when they are untagged, which is caused by a garbage collection being run at a point with more live (i.e., reachable) data.

Program	Lines	Memory (kb)			# GC			Time (s)		
		tag pairs	untag pairs	%	tag pairs	untag pairs	%	tag pairs	untag pairs	%
`vliw`	3676	1756	**1633**	7	43	**38**	12	5.7	**5.6**	2
`logic`	346	317	**290**	9	2565	**2302**	10	8.0	**7.1**	11
`tyan`	1018	2006	**1806**	10	357	**289**	19	7.4	**6.3**	15
`tsp`	493	7854	7854		11	11		6.5	6.5	
`DLX`	2836	2893	**2365**	18	3	**2**	33	7.4	**6.8**	8
`ratio`	619	1026	**950**	7	35	**27**	23	6.7	6.7	
`lexgen`	1318	3056	**2791**	9	168	**123**	27	6.6	**5.9**	11
`mlyacc`	7353	**2397**	2440	-2	361	**284**	21	6.8	**6.0**	12
`simple`	1052	1701	**1567**	8	9	9		6.9	**6.5**	6
`professor`	276	140	**139**	1	838	**575**	31	6.4	6.4	
`msort`	81	5929	**5423**	9	8	8		6.1	**5.6**	8
`kitlife`	230	64	**62**	3	2	2		5.6	5.6	

Fig. 8. Benchmark statistics.

The last three columns show the time (in seconds) for executing each benchmark with and without tagged pairs; improvements are shown in percentages. A comparison of the numbers with numbers obtained when garbage collection is disabled shows that, with respect to time, the untagging of pairs provides two kinds of savings. First, for programs that allocate many pairs in infinite regions, storing tag words for each pair is expensive. Second, for programs that use much memory, the effect that untagging of pairs has on the number of garbage collections significantly influences the overall time used for garbage collection.

6 Conclusion and Future Directions

We have presented a region type system, which guarantees that values of certain kinds (e.g., pairs) are stored in distinguished regions. Based on Cheney's algorithm for regions [9] and the guarantee provided by the region type system, we have presented an algorithm for partly tag-free garbage collection. Experimental results demonstrate that the algorithm improves memory usage and execution time, compared to the tag-based algorithm.

There are several directions for future work. First, it would be interesting to investigate if support for tag-free garbage collection of values other than pairs would have significant influence on memory usage and execution time.

Second, there are at least two ways the interaction between region inference and garbage collection can be improved. In particular, arranging that garbage collection can be initiated at arbitrary allocation points—instead of only at function entry points—may improve memory usage for some programs. Moreover, combining region inference with a multi-generational garbage collection scheme, where each region is associated with several generations, would make it possible

to garbage collect no longer reachable, newly allocated values without inspecting the entire heap.

References

1. Shail Aditya, Christine H. Flood, and James E. Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *LISP and Functional Programming*, pages 12–23, 1994.
2. Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.
3. 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.
4. Cristiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2), 2002.
5. C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.
6. Martin Elsman. Garbage collection safety for region-based memory management. In *Proceedings of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'03)*. ACM Press, January 2003. To appear.
7. Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proceedings of SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 165–176, June 1991.
8. Benjamin Goldberg and Michael Gloger. Polymorphic type reconstruction for garbage collection without tags. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 53–65. ACM Press, 1992.
9. Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM Press, June 2002.
10. David R. Hanson. A portable storage management system for the icon programming language. *Software—Practice and Experience*, 10:489–500, 1980.
11. Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In *Proceedings of the 4th International Workshop on Higher Order Operational Techniques in Semantics*, September 2000. Published in Volume 41(3) of the Electronic Notes in Theoretical Computer Science.
12. Greg Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, December 1995.
13. Greg Morrisett, David Tarditi, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. The TIL/ML compiler: Performance and safety through types. In *1996 Workshop on Compiler Support for Systems Software (WCSSS'96)*, February 1996.
14. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)*, pages 181–192, 1996.
15. Mads Tofte and Lars Birkedal. Unification and polymorphism in region inference. In *Proof, Language, and Interaction. Essays in Honour of Robin Milner*, May 2000.
16. Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical Report TR-2001-07, IT University of Copenhagen, October 2001.

17. Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report DIKU-report 93/15, Department of Computer Science, University of Copenhagen, 1993.
18. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
19. Andrew P. Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 1–11. ACM Press, 1994.
20. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.