Garbage-Collection Safety for Region-Based Type-Polymorphic Programs

Martin Elsman
Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
mael@di.ku.dk

Abstract

Region inference offers a mechanism to reduce (and sometimes entirely remove) the need for reference-tracing garbage collection by inferring where to insert allocation and deallocation instructions in a program at compile time. When the mechanism is combined with techniques for reference-tracing garbage collection, which is helpful in general to support programs with very dynamic memory behaviours, it turns out that region-inference is complementary to adding generations to a reference-tracing collector. However, region-inference and the associated region-representation analyses that make such a memory management strategy perform well in practice are complex, both from a theoretical point-of-view and from an implementation point-of-view.

In this paper, we demonstrate a soundness problem with existing theoretical developments, which have to do with ensuring that, even for higher-order polymorphic programs, no dangling-pointers appear during a reference-tracing collection. This problem has materialised as a practical soundness problem in a real implementation based on region inference. As a solution, we present a modified, yet simple, region type-system that captures garbage-collection effects, even for polymorphic higher-order code, and outline how region inference and region-representation analyses are adapted to the new type system. The new type system allows for associating simpler region type-schemes with functions, compared to original work, makes it possible to combine region-based memory management with partly tag-free reference-tracing (and generational) garbage-collection, and repairs previously derived work that is based on the erroneous published results.

CCS Concepts: • **Software and its engineering → Functional languages**; **Runtime environments**.

Keywords: region-inference, garbage-collection, Standard ML

ACM Reference Format:
Martin Elsman. 2022. Garbage-Collection Safety for Region-Based Type-Polymorphic Programs. In *DIKU Technical Report Series (DIKU Techreport 2022).* DIKU, DK-2100 Copenhagen, Denmark, 20 pages.

DIKU Techreport 2022, December 29, 2022, Copenhagen, Denmark
2022.

1 Introduction

Region-based memory management allows programmers to associate life-times of objects with so-called regions and to reason about how and when such regions are allocated and deallocated. Region-based memory management, as it is implemented for instance in Rust [3], can be a valuable tool for constructing certain kinds of critical systems, such as real-time embedded systems [37]. Region inference differs from explicit region-based memory management by taking a non-annotated program as input and producing a region-annotated program, including directives for allocating and deallocating regions [43]. The result is a programming paradigm where programmers can learn to write region-friendly code (by following certain patterns [44]) to obtain good space and time performance for critical parts of the program.

The region-based memory management scheme that we consider here is based on the stack discipline. Whenever e is some expression, region inference may decide to replace e with the term `letregion` ρ `in` e', where e' is the result of transforming the expression e, which includes annotating allocating expressions with particular region variables (e.g., ρ) specifying the region each value should be stored in. The semantics of the `letregion` term is first to allocate a region (initially an empty list of pages) on the region stack, bind the region to the region variable ρ, evaluate e', and, finally, deallocate the region bound to ρ (and its pages). The region type system allows regions to be passed to functions at run time (i.e., functions can be region-polymorphic) and to be captured in closures. The soundness of region inference ensures that a region is not deallocated as long as a value within it may be used by the remainder of the computation.

To remedy the problem that region inference does not always capture precisely the lifetime properties of objects, previous work has augmented the static inference scheme with more dynamic lifetime-based reference-tracing copying garbage collectors [16, 17, 24]. For such integrations of region-based memory management and reference-tracing garbage collection, care must be taken to rule out the possibility of deallocating regions with incoming pointers from live objects.

It turns out, however, that region inference (and the accompanying region typing rules) allows for so-called *dangling pointers*, which are pointers to objects that region inference has determined will not be needed by the remainder of the computation, yet are captured in objects (e.g., in closures) that escape a `letregion` construct and are live from a reference-tracing point-of-view.

Previous work attempt to rule out the possibility of dangling pointers by adjusting the region typing rules (and region inference) in such a way that the type of an object will mention all regions that the object may live in [13] (by enlarging the latent effect sets of certain function types). As a consequence, such an adjustment will capture the effect of a reference-tracing garbage collection appearing when control enters a function, for instance.

Unfortunately, it turns out that previous attempts at ruling out dangling pointers fail for certain programs that involve a combination of higher-order dead values and type polymorphism. From a theoretical point-of-view, the problem is that the region-typing rules, which form the basis of region inference, are not closed under type substitution, which is erroneously claimed by previous work [13, 17]. Moreover, as we shall see, this problem is not straightforward to overcome.

In practice, the erroneous theoretical results are exposed through the MLKit Standard ML compiler [44], which is a full Standard ML compiler that combines region inference and reference-tracing garbage collection. The MLKit compiles programs to efficient native machine code for Linux and macOS [14] and implements a number of techniques for refining the representations of regions [6, 43], including dividing regions into stack allocated (bounded) regions (also called *finite regions*) and heap allocated regions (also called *infinite regions*), which are the regions that are subject to reference-tracing garbage collections. As we shall see, based on the theoretical insights described above, it is possible to construct programs that fail during a reference-tracing collection due to the presence of dangling pointers at runtime.

Fortunately, it is possible to adjust the region type system to mitigate the problem and provide guarantees, also for higher-order type-polymorphic programs, that dangling pointers do not appear at runtime.

The contributions of this paper are the following:

1. We identify a safety problem with existing techniques for abandoning dangling-pointers at runtime, which serves as an assumption for combining region-inference with reference-tracing garbage collection.
2. We present a non-trivial modification to an existing region-based type system that rules out dangling pointers and allows for combining region-based memory management with reference-tracing (and even generational) garbage collection.
3. We describe how the modified type system affects region inference and the region-representation analyses that form the basis for a mature practical compiler infrastructure based on region inference.
4. We demonstrate that, in practice, the necessary modifications have little effect on performance and, in practice, affect only a small set of functions.

The paper is organised as follows. In the following section, we first give an informal example demonstrating a program for which dangling pointers will occur at runtime unless the region typing rules that form the basis of region inference are adjusted beyond previous suggestions. In this section, we also, informally, demonstrate how we may adjust the region typing rules further to completely eliminate the presence of dangling pointers.

In Section 3, we present a simplified, but formal, region type system for a language that serves as a target language for region inference. We present a number of properties of the type system, including region type soundness and the property that no dangling pointers are introduced during evaluation. In Section 4, we describe various aspects of the implementation, including how region inference is implemented for the system. We also give a number of examples demonstrating some non-trivial aspects of the system. In Section 5, we present a number of experimental results and evaluate the work. In Section 6, we describe related work, and in Section 7, we conclude.

2 The Problem

We now demonstrate the unsoundness problem that may occur when reference-tracing garbage collection is combined with higher-order functions and type-polymorphism. We present the problem in the context of a slightly modified region-based type system, compared to the original Tofte-Talpin region-type system, but emphasize that the unsoundness can be demonstrated also for the original system even if the typing rules are modified as described in [45, page 50] and [13], which aim at abandoning dangling pointers (but fail).

Consider the higher-order function for function composition, which has the following ML type-scheme:

```
val o : (γ → β) × (α → γ) → α → β
```

Here α, β, and γ are (implicitly quantified) type variables and o is an infix function that takes a pair of two functions as argument and returns a function as the result.

The region annotated version of the function o has the following region (and effect) type-scheme:

$$\begin{aligned}&\forall \epsilon\epsilon_0\epsilon_1\epsilon_2\rho_0\rho_1\rho_2\rho_3\alpha\beta\gamma. \\ &\quad((\gamma \xrightarrow{\epsilon_2.\emptyset} \beta, \rho_2) \times (\alpha \xrightarrow{\epsilon_1.\emptyset} \gamma, \rho_1), \rho_0) \\ &\quad\xrightarrow{\epsilon_0.\{\rho_0,\rho_3\}} \quad (\alpha \xrightarrow{\epsilon.\{\epsilon_1,\epsilon_2,\rho_1,\rho_2\}} \beta, \rho_3)\end{aligned} \tag{1}$$

```
fun run () : unit =
  let val h : unit -> unit =
        (op o)
        let val x = "oh" ^ "no"
        in (fn x => (), fn () => x)
        end
      val _ = work ()  (* trigger gc *)
  in h ()
  end
```

Figure 1. Problematic source program involving higher-order functions, type-polymorphism, and dead values.

Here ϵ, ϵ_0, ϵ_1, and ϵ_2 are effect variables and ρ_0, ρ_1, ρ_2, and ρ_3 are region variables. We see that function type constructors are annotated with so-called arrow effects, each of which is a set of atomic effects (effect variables and region variables) identified by an effect variable.[1] Moreover, type constructors for products ($\times$) and functions are annotated with region variables that indicate in which region a particular constructed value resides. The arrow effect $\epsilon_0.\{\rho_0, \rho_3\}$ expresses that when the function o is applied to a pair of functions, the pair, which resides in ρ_0 is deconstructed and a new closure is stored in region ρ_3. The arrow effect $\epsilon.\{\epsilon_1, \epsilon_2, \rho_1, \rho_2\}$, which appears on the arrow of the type of the resulting function, expresses that, when the function is applied, the two argument functions are accessed (ρ_2, ρ_1) and evaluated (ϵ_2, ϵ_1).

When a function such as o is applied, a particular instantiation of the function's type-scheme is described by a particular substitution that maps generic effect variables to arrow effects, generic region variables to region variables, and generic type variables to region-annotated types.

Consider now the problematic function run in Figure 1, which first creates a function h, thereby capturing a dead value in a closure, calls a function work (for the sake of triggering a reference-tracing collection), and finally calls the function h. Notice that the argument to the function o evaluates to a pair of functions for which the second function returns a pointer to an already allocated value ("ohno") and the first function will silently discard its argument.

Next, consider the region-annotated version of the function run, given in Figure 2(a). We see that region inference has determined that the closure bound to h will reside in the region ρ_3 and that the value bound to x will reside in the region ρ, which is deallocated after the function h is constructed.[2] The effect is that when the function work is called, which may perhaps trigger a reference-tracing collection, the value bound to h, which is live (and therefore part of the garbage-collection root set), will contain a pointer to an object that no longer exists.

Whereas the appearance of such dangling pointers is perfectly ok for a region-based memory management scheme that does not integrate with reference-tracing garbage collection (as long as the program itself does not dereference dangling pointers), a reference-tracing garbage collector will stumble over dangling pointers.

An alternative region-annotated version of the program appears in Figure 2(b). This region-annotated version of the program does not introduce dangling pointers at runtime as the region ρ live at least as long as the function h, which is enforced by ensuring that the type of the function h mentions the region ρ (in the arrow effect of the function type).

We now describe, informally, the mechanism that enforces region inference to assign the type unit $\xrightarrow{\epsilon.\{\rho_1,\rho_2,\rho\}}$ unit to the function h. First, notice that the type of h is the result type of an instance of the type scheme for the function o and that we must somehow capture, in the type scheme for o, that the type instance for the type variable γ specifies values that live in the region ρ. We can capture this property by giving o the following type scheme:

$$\begin{array}{l}\forall \epsilon\epsilon_0\epsilon_1\epsilon_2\epsilon'\rho_0\rho_1\rho_2\rho_3\alpha\beta(\gamma : \epsilon'.\emptyset). \\ \quad ((\gamma \xrightarrow{\epsilon_2.\emptyset} \beta, \rho_2) \times (\alpha \xrightarrow{\epsilon_1.\emptyset} \gamma, \rho_1), \rho_0) \\ \quad \xrightarrow{\epsilon_0.\{\rho_0,\rho_3\}} \quad (\alpha \xrightarrow{\epsilon.\{\epsilon_1,\epsilon_2,\epsilon',\rho_1,\rho_2\}} \beta, \rho_3)\end{array} \tag{2}$$

Compared to (1), the modified type scheme expresses a relationship between the type variable γ, through the *type variable descriptor* $\gamma : \epsilon'.\emptyset$, and the effect of the resulting function, which can be used to establish that regions appearing in the type of the instantiated type for γ must appear in the effect identified by the effect variable ϵ'. In the theoretical development presented in the following sections, this establishment will be implemented as part of the instance-of relation between region type-schemes and region-annotated types. Moreover, the typing rule for functions will ensure that type variables that appear in the type of a free variable occurring in the body of the function are associated with effect variables that are added to the arrow effect of the function type.

An alternative sound type scheme for o is the following:

$$\begin{array}{l}\forall \epsilon\epsilon_0\epsilon_1\epsilon_2\rho_0\rho_1\rho_2\rho_3\alpha\beta(\gamma : \epsilon.\{\epsilon_1, \epsilon_2, \rho_1, \rho_2\}). \\ \quad ((\gamma \xrightarrow{\epsilon_2.\emptyset} \beta, \rho_2) \times (\alpha \xrightarrow{\epsilon_1.\emptyset} \gamma, \rho_1), \rho_0) \\ \quad \xrightarrow{\epsilon_0.\{\rho_0,\rho_3\}} \quad (\alpha \xrightarrow{\epsilon.\{\epsilon_1,\epsilon_2,\rho_1,\rho_2\}} \beta, \rho_3)\end{array} \tag{3}$$

Compared to (2), the alternative type scheme identifies the arrow effects associated with the result function type and

[1] As described in details later, allowing arrow effects to be identified by effect variables (so-called *effect-handles*) enables the possibility that effects may grow by applying effect substitutions (which map effect variables to arrow effects).

[2] Notice that string concatenation (^) takes, besides the two argument strings, the region (ρ) into which the result is allocated.

```
fun run () : unit =
  letregion ρ1,ρ2,ρ3 in

    let val h : (unit -ϵ.{ρ1,ρ2}-> unit, ρ3) =
          letregion ρ,ρ0 in
            (op o [ρ3])
            let val x = op ^ [ρ] ("oh","no")
            in (fn at ρ1 x => (),
                fn at ρ2 () => x) at ρ0
            end
          end
        val _ = work ()  (* trigger gc *)
    in h ()
    end
  end
```

(a)

```
fun run () : unit =
  letregion ρ,ρ1,ρ2,ρ3 in

    let val h : (unit -ϵ.{ρ1,ρ2,ρ}-> unit, ρ3) =
          letregion ρ0 in
            (op o [ρ3])
            let val x = op ^ [ρ] ("oh","no")
            in (fn at ρ1 x => (),
                fn at ρ2 () => x) at ρ0
            end
          end
        val _ = work ()  (* trigger gc *)
    in h ()
    end
  end
```

(b)

Figure 2. An unsound region-annotated program (a) and an alternative sound region-annotated program (b).

the type variable γ, which is fine for the function o. Such an identification, which is perfectly sound, can be problematic (i.e., cause larger live ranges of regions), however, for type schemes with multiple type variables occurring free in the types of free identifiers of a function. On the positive side, however, the alternative type scheme can be expressed without introducing new *secondary* effect variables,[3] which can be problematic for region inference.

Both of the above type schemes are sound candidates for providing a type scheme for the composition function o. And indeed, both type schemes are accepted by the GC-safe region type system that we present in the next section. Distinguishing between the type system and the inference algorithm is vital here as it provides us with important implementation flexibility. We will return to the details of the inference algorithm and implementation choices in Section 4.

3 A GC-Safe Region Type System

In this section, we present a type system that provides us with the necessary guarantees for integrating region inference and reference-tracing garbage collection. Compared to the Tofte-Talpin type system [46], the type system that we present ensures that no dangling pointers are introduced during evaluation even for programs that involve higher-order type-polymorphic functions.

In the remainder of this section, we present a formal treatment for a small ML-like intermediate language extended with region annotations.

3.1 Regions and Effects

We assume a denumerably infinite set of *program variables*, ranged over by x and f. We also assume a denumerably infinite set of *region variables*, ranged over by ρ. Moreover, we assume a denumerably infinite set of *effect variables*, ranged over by ϵ. An *atomic effect*, ranged over by η, is either a region variable or an effect variable, and an *effect*, ranged over by φ, is a set of atomic effects. An *arrow effect*, written $\epsilon.\varphi$, and ranged over by ν, is a pair of an effect variable and an effect. Finally, we assume a denumerably infinite set of *type variables*, ranged over by α.

A *type variable context*, ranged over by Ω (or Δ), is a finite map from type variables to arrow effects. When M and M' are two finite maps, we write $M + M'$ to denote the map with domain $\mathrm{dom}(M) \cup \mathrm{dom}(M')$ and values $(M + M')(x) = M'(x),$ if $x \in \mathrm{dom}(M')$ and $M(x),$ otherwise.

For simplicity, we do not distinguish between put- and get-effects in the formal treatment of effects. However, for reasons that we shall make clear later, function types are annotated with arrow effects and not only with effects.

3.2 Types and Type Schemes

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

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

For type schemes of the form $\forall \vec{\rho}\vec{\epsilon}.\sigma$, the region variables $\vec{\rho}$ and the effect variables $\vec{\epsilon}$ are considered *bound* in σ. Moreover, in type schemes of the form $\forall \Delta.\tau$, the type variables in $\mathrm{dom}(\Delta)$ are considered bound in τ. Type schemes are considered identical up to renaming of bound variables.

[3] A secondary effect variable is an effect variable that does not appear syntactically as a handle on an arrow type constructor anywhere in the type-annotated version of the program.

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.

A type and place μ (or type τ) is *well-formed* with respect to a type variable context Ω, if the sentence $\Omega \vdash \mu$ (or $\Omega \vdash \tau$) can be derived from the following rules:

Well-formed types $\boxed{\Omega \vdash \mu}$

$$\frac{\alpha \in \mathrm{dom}(\Omega)}{\Omega \vdash \alpha} \qquad \frac{}{\Omega \vdash \mathtt{int}}$$

$$\frac{\Omega \vdash \mu_1 \quad \Omega \vdash \mu_2}{\Omega \vdash (\mu_1 \times \mu_2, \rho)} \qquad \frac{\Omega \vdash \mu_1 \quad \Omega \vdash \mu_2}{\Omega \vdash (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho)}$$

Further, a type scheme and place π (or type scheme σ) is *well-formed* with respect to a type variable context Ω, if the sentence $\Omega \vdash \pi$ (or $\Omega \vdash \sigma$) can be derived from the following rules:

Well-formed type schemes $\boxed{\Omega \vdash \pi}$

$$\frac{\Omega \vdash (\sigma, \rho)}{\Omega \vdash (\forall \vec{\rho}\vec{\epsilon}.\sigma, \rho)}$$

$$\frac{\Omega + \Delta \vdash (\tau, \rho) \quad \mathrm{dom}(\Delta) \cap \mathrm{dom}(\Omega) = \emptyset}{\Omega \vdash (\forall \Delta.\tau, \rho)}$$

Before we define the notion of substitution, we define a notion of *type containment*, which expresses that a given type τ (and place μ) is contained in an effect φ, under the assumption of a type variable context Ω. The relation is written $\Omega \vdash \mu : \varphi$ and is defined according to the following rules:

Type containment $\boxed{\Omega \vdash \mu : \varphi}$

$$\frac{\Omega \vdash \mu_1 : \varphi \quad \Omega \vdash \mu_2 : \varphi \quad \rho \in \varphi}{\Omega \vdash (\mu_1 \times \mu_2, \rho) : \varphi}$$

$$\frac{\Omega \vdash \mu_1 : \varphi \quad \Omega \vdash \mu_2 : \varphi \quad \varphi_0 \subseteq \varphi \quad \{\rho, \epsilon\} \subseteq \varphi}{\Omega \vdash (\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2, \rho) : \varphi}$$

$$\frac{}{\Omega \vdash \mathtt{int} : \varphi} \qquad \frac{\mathrm{frev}(\Omega(\alpha)) \subseteq \varphi}{\Omega \vdash \alpha : \varphi}$$

Containment is extended to type schemes as follows:

Type scheme containment $\boxed{\Omega \vdash \pi : \varphi}$

$$\frac{\begin{array}{c}\Omega \vdash \sigma : \varphi \\ \rho \in \varphi \quad \{\vec{\rho}\vec{\epsilon}\} \cap \mathrm{frev}(\Omega, \rho) = \emptyset\end{array}}{\Omega \vdash (\forall \vec{\rho}\vec{\epsilon}.\sigma, \rho) : \varphi \setminus \{\vec{\rho}\vec{\epsilon}\}}$$

$$\frac{\Omega + \Delta \vdash (\tau, \rho) : \varphi \quad \mathrm{dom}(\Delta) \cap \mathrm{dom}(\Omega) = \emptyset}{\Omega \vdash (\forall \Delta.\tau, \rho) : \varphi}$$

Proposition 1 (Containment Implies Well-formedness). *Assume o is one of μ or π. If $\Omega \vdash o : \varphi$ then $\Omega \vdash o$.*

Proof. By simple induction over the structure of o. □

Both well-formedness and containability features context extensibility properties. Assume o is one of μ or π and $\mathrm{dom}(\Omega) \cap \mathrm{dom}(\Delta) = \emptyset$. If $\Omega \vdash o : \varphi$ then $\Omega + \Delta \vdash o : \varphi$. Moreover, if $\Omega \vdash o$ then $\Omega + \Delta \vdash o$.

It is also straightforward to demonstrate an effect extensibility property for type containment stating that, when o is one of μ or π, if $\Omega \vdash o : \varphi$ and $\varphi \subseteq \varphi'$ then $\Omega \vdash o : \varphi'$.

Finally, the following effect containment property holds:

Proposition 2 (Containment). *Assume o is one of μ or π. If $\Omega \vdash o : \varphi$ then* $\mathrm{frev}(o) \subseteq \varphi$.

Proof. By simple induction over the structure of o. □

3.3 Substitutions

A *substitution* (S) is a triple $(S^{\mathrm{t}}, S^{\mathrm{r}}, S^{\mathrm{e}})$, where S^{t} is a *type substitution*, a finite map from type variables to type and places, S^{r} is a *region substitution*, a finite map from region variables to region variables, and S^{e} is an *effect substitution*, a finite map from effect variables to arrow effects. The effect of applying a substitution to a particular object is to carry out the three substitutions simultaneously to the three kinds of variables in the object (possibly by renaming of bound variables within the object to avoid capture) and acting as the identity outside of its domain. For effect sets and arrow effects, substitution is defined as follows [42], assuming $S = (S^{\mathrm{t}}, S^{\mathrm{r}}, S^{\mathrm{e}})$:

$$\begin{array}{rcl} S(\varphi) & = & \{S^{\mathrm{r}}(\rho) \mid \rho \in \varphi\} \cup \\ & & \{\eta \mid \exists \epsilon.\epsilon \in \varphi \wedge \eta \in \mathrm{frev}(S^{\mathrm{e}}(\epsilon))\} \\ & & \\ S(\epsilon.\varphi) & = & \epsilon'.(\varphi' \cup S(\varphi)),\ \text{where } S^{\mathrm{e}}(\epsilon) = \epsilon'.\varphi' \end{array}$$

Notice in particular, that when a substitution is applied to an effect φ, the result is also an effect.

Applying a substitution S to a type variable context Δ is defined only if $\mathrm{dom}(S) \cap \mathrm{dom}(\Delta) = \emptyset$, in which case it is defined as follows:

$$S(\{\alpha_1 : \nu_1, \cdots, \alpha_n : \nu_n\}) = \{\alpha_1 : S(\nu_1), \cdots, \alpha_n : S(\nu_n)\}$$

For type schemes and for type-schemes and places, substitution is defined as follows, assuming that bound variables in type schemes have been renamed to avoid capture:

$$\begin{array}{rcl} S(\forall \vec{\rho}\vec{\epsilon}.\sigma) & = & \forall \vec{\rho}\vec{\epsilon}.S(\sigma) \\ S(\forall \Delta.\tau) & = & \forall S(\Delta).S(\tau) \\ S(\sigma, \rho) & = & (S(\sigma), S(\rho)) \end{array}$$

It turns out that substitution is a monotone operation with respect to effects:

Proposition 3 (Substitution Effect Monotonicity). *If $\varphi \subseteq \varphi'$ then $S(\varphi) \subseteq S(\varphi')$, for any substitution S and effects φ and φ'.*

Proof. Follows immediately from the definition of substitution on effects. □

Another property that holds, which we shall call the *arrow-effect-substitution interchange property*, is that for any substitution S and arrow effect $\epsilon.\varphi$, we have $\mathrm{frev}(S(\epsilon.\varphi)) = S(\{\epsilon\} \cup \varphi)$.

If $S = (S^{\mathrm{t}}, S^{\mathrm{r}}, S^{\mathrm{e}})$, we call S a *region-effect substitution* if $\mathrm{dom}(S^{\mathrm{t}}) = \emptyset$. Type containment is closed under region-effect substitutions:

Proposition 4 (Region-Effect Substitution Closedness). *Assume o is one of μ or π. If $\Omega \vdash o : \varphi$ and S is a region-effect substitution then $S(\Omega) \vdash S(o) : S(\varphi)$.*

Proof. Straightforward induction over the structure of o. □

For type containment to be closed under type substitutions, a substitution coverage requirement is needed. A type substitution S^{t} is *covered* by a type variable context Ω, through another type variable context Δ, written $\Omega \vdash S^{\mathrm{t}} : \Delta$, if $\mathrm{dom}(S^{\mathrm{t}}) = \mathrm{dom}(\Delta)$ and, for all $\alpha \in \mathrm{dom}(S^{\mathrm{t}})$, we have $\Omega \vdash S^{\mathrm{t}}(\alpha) : \mathrm{frev}(\Delta(\alpha))$.

In connection with the notion of instantiation, which we shall define shortly, it is the notion of substitution coverage that ensures that the arrow effect associated with a bound type variable captures the free region and effect variables of the types instantiated for the type variable (which also holds transitively via the type containment relation.)

Proposition 5 (Type Substitution Closedness). *Assume o is one of μ or π. If $\Omega + \Delta \vdash o : \varphi$ and $\Omega \vdash S : \Delta$ then $\Omega \vdash S(o) : \varphi$.*

Proof. By induction over the structure of o. The interesting case is the case for $\mu = \alpha$ for some type variable α. There are now two cases. We first consider the case where $\alpha \in \mathrm{dom}(S)$. From the definition of coverage, we have $\langle 1 \rangle$ $\Omega \vdash S(\alpha) : \mathrm{frev}(\Delta(\alpha))$ and $\mathrm{dom}(S) = \mathrm{dom}(\Delta)$. Moreover, from assumptions we have $\Omega + \Delta \vdash \alpha : \varphi$, thus, from the definition of containment, we have $\mathrm{frev}((\Omega + \Delta)(\alpha)) \subseteq \varphi$ and thus $\langle 2 \rangle$ $\mathrm{frev}(\Delta(\alpha)) \subseteq \varphi$. From the extensibility property of type containment and from $\langle 1 \rangle$ and $\langle 2 \rangle$, we have $\Omega \vdash S(\alpha) : \varphi$, as required. For the second case where $\alpha \notin \mathrm{dom}(S)$, we have $S(\alpha) = \alpha$. It follows from the definition of coverage that $\alpha \notin \mathrm{dom}(\Delta)$, which leads us to conclude, based on the assumptions and the definition of type containment, that $\Omega \vdash S(\alpha) : \varphi$, as required. □

3.4 Instantiation

Given a type variable context Ω and a type scheme $\sigma = \forall\Delta.\tau'$ such that $\Omega \vdash \sigma$, a type τ is an *instance of* σ via a type substitution S^{t}, written $\Omega \vdash \sigma \geq \tau$ via S^{t}, if

1. $\Omega \vdash S^{\mathrm{t}} : \Delta$
2. $S^{\mathrm{t}}(\tau') = \tau$

Given a type variable context Ω and a type scheme $\sigma = \forall\vec{\rho}\vec{\epsilon}.\sigma'$ such that $\Omega \vdash \sigma$, a type τ is an *instance of* σ via a substitution $S = (S^{\mathrm{t}}, S^{\mathrm{r}}, S^{\mathrm{e}})$, written $\Omega \vdash \sigma \geq \tau$ via S, if

1. $\mathrm{dom}(S^{\mathrm{r}}) = \{\vec{\rho}\}$ and $\mathrm{dom}(S^{\mathrm{e}}) = \{\vec{\epsilon}\}$
2. $\Omega \vdash S^{\mathrm{e}}(S^{\mathrm{r}}(\sigma')) \geq \tau$ via S^{t}

When we are interested in only the region instance list, we write $\Omega \vdash \sigma \geq \tau$ via $\vec{\rho}$ to mean there exists a substitution $S = (S^{\mathrm{t}}, S^{\mathrm{r}}, S^{\mathrm{e}})$ such that $\Omega \vdash \sigma \geq \tau$ via S and $\mathrm{rng}(S^{\mathrm{r}}) = \{\vec{\rho}\}$.

It holds that if $\Omega \vdash \sigma \geq \tau$ via $\vec{\rho}$, for some σ, τ, and $\vec{\rho}$, and S is a region-effect substitution, then $S(\Omega) \vdash S(\sigma) \geq S(\tau)$ via $S(\vec{\rho})$. Moreover, if $\Omega + \Delta \vdash \sigma \geq \tau$ via $\vec{\rho}$, for some Ω, Δ, σ, τ, and $\vec{\rho}$, if $\Omega \vdash S : \Delta$, then $\Omega \vdash S(\sigma) \geq S(\tau)$ via $\vec{\rho}$. These properties are corollaries of the following, more general, propositions:

Proposition 6 (Instantiation Closed Under Region-Effect Substitution). *If S is a region-effect substitution and $\Omega \vdash \sigma \geq \tau$ via S' then $S(\Omega) \vdash S(\sigma) \geq S(\tau)$ via S'', where $S'' = (S \circ S') \downarrow \mathrm{dom}(S')$.*

Proof. We first consider the case where $\sigma = \forall\Delta.\tau'$. From the definition of instantiation, we have $\langle 1 \rangle$ $S'(\tau') = \tau$ and $\langle 2 \rangle$ $\Omega \vdash S' : \Delta$, and, thus, $\langle 3 \rangle$ $\mathrm{dom}(S') = \mathrm{dom}(\Delta)$. Because S is a region-effect substitution, we have $\langle 4 \rangle$ $S(\sigma) = \forall S(\Delta).S(\tau')$ and $\langle 5 \rangle$ $\mathrm{dom}(\Delta) = \mathrm{dom}(S(\Delta))$ and $\langle 6 \rangle$ $\mathrm{dom}(\Delta) \cap \mathrm{fv}(\mathrm{rng}(S)) = \emptyset$. Now, let $S'' = ((S \circ S') \downarrow \mathrm{dom}(S'))$. It follows that we have $\langle 7 \rangle$ $\mathrm{dom}(S'') = \mathrm{dom}(S(\Delta))$. We also have $S(S'(\tau')) = S(\tau)$ from $\langle 1 \rangle$ and $\langle 8 \rangle$ $S \circ S' = S'' \circ S$ because of $\langle 6 \rangle$ and $\langle 3 \rangle$. It follows that we have $\langle 9 \rangle$ $S''(S(\tau')) = S(\tau)$. We now need to show $S(\Omega) \vdash S'' : S(\Delta)$. From $\langle 2 \rangle$ and the definition of substitution coverage, we have $\langle 10 \rangle$ $\Omega \vdash S'(\alpha) : \mathrm{frev}(\Delta(\alpha))$, for all $\alpha \in \mathrm{dom}(S')$. From Proposition 4 and $\langle 10 \rangle$, we have $S(\Omega) \vdash S(S'(\alpha)) : S(\mathrm{frev}(\Delta(\alpha)))$ and thus, from $\langle 8 \rangle$ and because $\mathrm{dom}(S') = \mathrm{dom}(S'')$ follows from the definition of S'', we have $\langle 11 \rangle$ $S(\Omega) \vdash S''(\alpha) : \mathrm{frev}(S(\Delta)(\alpha))$, for all $\alpha \in \mathrm{dom}(S'')$. It follows from $\langle 11 \rangle$ that we have $\langle 12 \rangle$ $S(\Omega) \vdash S'' : S(\Delta)$. Now, from the definition of instantiation and from $\langle 9 \rangle$ and $\langle 12 \rangle$, we have $S(\Omega) \vdash S(\sigma) \geq S(\tau')$ via S'', as required. □

Proposition 7 (Instantiation Closed Under Type Substitution). *If $\Omega + \Delta \vdash \sigma \geq \tau$ via S' and $\Omega \vdash S : \Delta$ then $\Omega \vdash S(\sigma) \geq S(\tau)$ via S'', where $S'' = (S \circ S') \downarrow \mathrm{dom}(S')$.*

A *type environment* (Γ) maps program variables to type schemes and places. Given a type variable context Ω, a type environment Γ is *well-formed in* Ω, written $\Omega \vdash \Gamma$, if $\Omega \vdash \pi$, for all type schemes and places $\pi \in \mathrm{rng}(\Gamma)$.

3.5 The Role of Arrow Effects

We emphasize that function types are annotated with arrow effects $\epsilon.\varphi$ and not only with effects φ; with arrow effects, we can allow for effects to grow and we can make sure that if a non-region-annotated type is given two distinct region-annotations, then there exists a substitution, a *unifier*, that, when applied to the two types, will make the two resulting region-annotated types equal. This property is essential for the applied unification-based region inference algorithm [41], which we shall discuss further later.

Moreover, notice that, for each object that we deal with, when an effect variable appears free in the object, it is made explicit what effect it denotes, except when an effect variable appears free in an effect; in this case, however, we know that, due to an assumed transitivity of effects, the effect already includes the effect denoted by the included effect variable. It is for this reason that we annotate quantified type variables with arrow effects and not only with effect variables; we often (e.g., in the typing rules) need to know what effect the effect variable denotes. An alternative would be, in the typing rules, to keep track of the denotation of effect variables in an external *effect basis*, similarly to how effects are treated in the description of region inference [41, 42]; making the effect basis explicit makes it straightforward to formulate certain well-formedness and consistency constraints on the effect variables and their denotations in the rules. For instance, if $\epsilon.\varphi$ and $\epsilon'.\varphi'$ are two arrow effects appearing in the derivation of some judgement, then $\epsilon = \epsilon'$ implies $\varphi = \varphi'$ (the basis is *functional*) and $\epsilon' \in \varphi$ implies $\varphi' \subseteq \varphi$ (the basis is *transitive*).

3.6 Terms

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

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

Values include unboxed integers (d), pairs, ordinary closures, and recursive function closures (which may also take regions as parameters). All values, except integers, are boxed and associated with distinguished regions. An expression can be a value, a variable, a `let`-expression, a function application, a lambda-expression, a `letregion`-construct, a recursive function binding, an application of a recursive function to a list of region parameters, a pair-construct, and a pair-projection expression. Notice that allocating expressions are annotated with an `at`-specifier, which specifies in which region the value should be allocated. Notice also that expressions may contain values. A program does not contain values initially. During evaluation, however, variables in the program may be substituted with values, which is captured precisely by the small-step dynamic semantics that we shall define later.

In expressions of the forms `let` $x = e_1$ `in` e and $\lambda x.e$ `at` ρ, the variable x is *bound* in e. In expressions of the form `fun` f $[\vec{\rho}]$ $x = e$ `at` ρ, the variables f, $\vec{\rho}$, and x are *bound* in e. Similarly for values. In expressions `letregion` ρ `in` e, the variable ρ is *bound* in e. As usual, we identify terms up to renaming of bound variables. The *free (program) variables* of some expression (or value) e is written $\mathrm{fpv}(e)$.

3.7 Value Containment and GC Safety

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 [46]. 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 [13].

First, we define a notion of *value containment*; all values in an expression e are contained in a set of regions φ, if the sentence $\varphi \models_{\mathrm{v}} e$ is derivable from the rules in Figure 3. It is straightforward to demonstrate that if $\varphi \models_{\mathrm{v}} e$ and $\varphi \subseteq \varphi'$ then $\varphi' \models_{\mathrm{v}} e$ (*value containment extensibility*). Moreover, for any substitution S, it follows that $S(\varphi) \models_{\mathrm{v}} S(e)$. Finally, if $\varphi \models_{\mathrm{v}} e$ and $\varphi \models v$ then $\varphi \models_{\mathrm{v}} e[v/x]$ (*value containment substitution*).

We now introduce a *GC-Safety relation 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 [45, page 50] and is parameterised over a type variable context Ω, an environment Γ, a function body e, a set of function parameters X, and the type scheme and place π of the function:

$$\begin{array}{rcl} G(\Omega, \Gamma, e, X, \pi) & = & \mathrm{frv}(\pi) \models_{\mathrm{v}} e\ \wedge \\ & & \forall y \in \mathrm{fpv}(e) \setminus X. \\ & & \quad \Omega \vdash \Gamma(y) : \mathrm{frev}(\pi) \end{array} \tag{4}$$

The garbage-collection safety relation is closed under region-effect substitution:

Proposition 8 (GC-Safety Relation Closed Under Region–Effect Substitution)**.** *If $G(\Omega, \Gamma, e, X, \pi)$ and S is a region-effect substitution then $G(S(\Omega), S(\Gamma), S(e), X, S(\pi))$.*

Proof. Follows immediately from the definition of garbage-collection safety, the property that value-containment is closed under substitution, and Proposition 4. □

The garbage-collection safety relation is also closed under type substitution, assuming that the substitution is properly covered:

Proposition 9 (GC-Safety Relation Closed Under Type Substitution)**.** *Assume $\Omega \vdash S : \Delta$. If $G(\Omega + \Delta, \Gamma, e, X, \pi)$ then $G(\Omega, S(\Gamma), e, X, S(\pi))$.*

Proof. From assumptions and because $\mathrm{frv}(S(\pi)) \supseteq S(\mathrm{frv}(\pi))$, for any substitution S, we have, because value containment is closed under substitution and due to value containment extensibility, that

$$\mathrm{frv}(S(\pi)) \models_{\mathrm{v}} S(e)$$

Because $\mathrm{fpv}(S(e)) = \mathrm{fpv}(e)$, it remains to be shown that

$$\begin{array}{l} \forall y \in \mathrm{fpv}(e) \setminus X. \\ \quad \Omega \vdash S(\Gamma(y)) : \mathrm{frev}(S(\pi)) \end{array}$$

From assumptions, we have that for all $y \in \mathrm{fpv}(e) \setminus X$,

$$\Omega + \Delta \vdash \Gamma(y) : \mathrm{frev}(\pi)$$

From Proposition 5 and assumptions, we have

$$\Omega \vdash S(\Gamma(y)) : S(\mathrm{frev}(\pi))$$

Values $\boxed{\varphi \models v}$

$$\varphi \models d \qquad \frac{\varphi \models_{\mathrm{v}} e \quad \rho \in \varphi}{\varphi \models \langle \lambda x.e \rangle^{\rho}} \qquad \frac{\varphi \models v_1 \quad \varphi \models v_2 \quad \rho \in \varphi}{\varphi \models \langle v_1, v_2 \rangle^{\rho}} \qquad \frac{\rho \in \varphi \quad \varphi \models_{\mathrm{v}} e \quad \{\vec{\rho}\} \cap \varphi = \emptyset}{\varphi \models \langle \mathtt{fun}\ f\ [\vec{\rho}]\ x = e \rangle^{\rho}}$$

Expressions $\boxed{\varphi \models_{\mathrm{v}} e}$

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

$$\frac{\varphi \models_{\mathrm{v}} e \quad \{\vec{\rho}\} \cap \varphi = \emptyset}{\varphi \models_{\mathrm{v}} \mathtt{fun}\ f\ [\vec{\rho}]\ x = e\ \mathtt{at}\ \rho} \qquad \frac{\varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{v}} e\ [\vec{\rho}]\ \mathtt{at}\ \rho} \qquad \frac{\varphi \models_{\mathrm{v}} e_1 \quad \varphi \models_{\mathrm{v}} e_2}{\varphi \models_{\mathrm{v}} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2} \qquad \frac{\rho \notin \varphi \quad \varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{v}} \mathtt{letregion}\ \rho\ \mathtt{in}\ e}$$

Figure 3. Value containment.

Now, because $\mathrm{frev}(S(\pi)) \supseteq S(\mathrm{frev}(\pi))$ and because of type-containment effect-extensibility, we have

$$\Omega \vdash S(\Gamma(y)) : \mathrm{frev}(S(\pi))$$

as required. □

Proposition 10 (GC-Safety Relation Closed Under Value Substitution)**.** *If $x \notin X$ and $G(\Omega, \Gamma + \{x : \pi\}, e, X, \pi')$ and* $\mathrm{frv}(\pi) \models v$ *and* $\mathrm{fpv}(v) = \emptyset$ *then* $G(\Omega, \Gamma, e[v/x], X, \pi')$.

Proof. From assumptions and (4), we have

$$\mathrm{frv}(\pi') \models_{\mathrm{v}} e \tag{5}$$

$$\forall y \in \mathrm{fpv}(e) \setminus X.\Omega \vdash (\Gamma + \{x : \pi\})(y) : \mathrm{frev}(\pi') \tag{6}$$

First, assume $x \in \mathrm{fpv}(e)$. Because $x \notin X$, by choosing x for y, we have from (6) that $\Omega \vdash \pi : \mathrm{frev}(\pi')$. It follows from Proposition 2 that $\mathrm{frev}(\pi) \subseteq \mathrm{frev}(\pi')$ and, thus

$$\mathrm{frv}(\pi) \subseteq \mathrm{frv}(\pi') \tag{7}$$

It follows from assumption, (7), and the value containment extensibility property that we have

$$\mathrm{frv}(\pi') \models v \tag{8}$$

Now, from (5), (8), and the value containment substitution property, we have

$$\mathrm{frv}(\pi') \models_{\mathrm{v}} e[v/x] \tag{9}$$

We also have from (6) and because fpv(v) = $\emptyset$ that

$$\forall y \in \mathrm{fpv}(e[v/x]) \setminus X.\Omega \vdash \Gamma(y) : \mathrm{frev}(\pi') \tag{10}$$

From (4), (9), and (10), we have $G(\Omega, \Gamma, e[v/x], X, \pi')$, as required. □

3.8 Typing Rules

The typing rules for values and expressions are mutually dependent and are shown in Figure 4. The typing rules for values allow inference of sentences of the form $\vdash v : \pi$, which states that "the value v has type scheme and place π". The typing rules for expressions allow inference of sentences of the form $\Omega, \Gamma \vdash e : \pi, \varphi$, which states that "in the type variable context Ω and in the type environment Γ, the expression e has type scheme and place π and effect φ.

There are a number of observations to be made about the typing rules. First, notice that the typing of values is specified without a variable environment, which, implicitly, specifies that well-typed values must be closed with respect to program variables. Moreover, values have no effect. Notice also that the typing rules for closures and for region- and effect-polymorphic function values specify that values within function bodies are contained in regions that appear in the type schemes for the functions (ensured using the value-containment judgement).

For lambda-expressions and region- and effect-polymorphic function expressions, gc-safety properties are specified using the gc-safety relation, which generalises the containment conditions specified in the corresponding value typing rules. Moreover, notice that there are two rules for typing region- and effect-polymorphic function expressions (and values), one that supports recursion (and even region- and effect-polymorphic recursion) and one that supports parameterisation of effects that are associated with quantified type variables. The reason for this duplication is that we must be careful that polymorphic recursion only quantify over region and effect variables that do not appear in type variable contexts that specify quantified type variables in the type scheme of the function.

For simplicity, the typing rule for `let`-bindings does not allow for generalisation.

3.9 Typing Properties

The typing rules are closed under region-effect substitution.

Proposition 11 (Typing Closed Under Region-Effect Substitution)**.** *Assume S is a region-effect substitution.*

1. *If* $\Omega, \Gamma \vdash e : \pi, \varphi$ *then* $S(\Omega), S(\Gamma) \vdash S(e) : S(\pi), S(\varphi)$.
2. *If* $\vdash v : \pi$ *then* $\vdash S(v) : S(\pi)$.

Proof. By simultaneous induction over the derivation of $\Omega, \Gamma \vdash e : \pi, \varphi$ and the derivation of $\vdash v : \pi$. □

The typing rules are also closed under type substitution provided the substitution is properly covered:

Values $\boxed{\vdash v : \pi}$

$$\frac{}{\vdash d : \mathtt{int}} \qquad \frac{\begin{array}{c}\{\}, \{x : \mu_1\} \vdash e : \mu_2, \varphi \\ \mu = (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \quad \vdash \mu \quad \mathrm{frv}(\mu) \models_{\mathrm{v}} e\end{array}}{\vdash \langle \lambda x.e \rangle^{\rho} : \mu}\;[\textsc{TvLam}] \qquad \frac{\vdash v_1 : \mu_1 \quad \vdash v_2 : \mu_2}{\vdash \langle v_1, v_2 \rangle^{\rho} : (\mu_1 \times \mu_2, \rho)}\;[\textsc{TvPair}]$$

$$\frac{\begin{array}{c}\Delta, \{x : \mu_1\} \vdash e : \mu_2, \varphi \quad \vdash \pi \\ \mathrm{frev}(\vec{\rho}\vec{\epsilon}) \cap \{\rho\} = \emptyset \quad \pi = (\forall \vec{\rho}\vec{\epsilon}\Delta.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \quad \mathrm{frv}(\pi) \models_{\mathrm{v}} e\end{array}}{\vdash \langle \mathtt{fun}\ f\ [\vec{\rho}]\ x = e \rangle^{\rho} : \pi}\;[\textsc{TvFun}]$$

$$\frac{\begin{array}{c}\Delta, \{f : (\forall \vec{\rho}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), x : \mu_1\} \vdash e : \mu_2, \varphi \quad \vdash \pi \\ \mathrm{frev}(\vec{\rho}\vec{\epsilon}) \cap \mathrm{frev}(\Delta) = \emptyset \quad \mathrm{frev}(\vec{\rho}\vec{\epsilon}) \cap \{\rho\} = \emptyset \quad \pi = (\forall \vec{\rho}\vec{\epsilon}\Delta.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \quad \mathrm{frv}(\pi) \models_{\mathrm{v}} e\end{array}}{\vdash \langle \mathtt{fun}\ f\ [\vec{\rho}]\ x = e \rangle^{\rho} : \pi}\;[\textsc{TvRec}]$$

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

$$\frac{\vdash v : \pi}{\Omega, \Gamma \vdash v : \pi, \emptyset}\;[\textsc{TeVal}] \qquad \frac{\varphi' \supseteq \varphi \quad \Omega, \Gamma \vdash e : \pi, \varphi}{\Omega, \Gamma \vdash e : \pi, \varphi'}\;[\textsc{TeSub}] \qquad \frac{\Gamma(x) = \pi}{\Omega, \Gamma \vdash x : \pi, \emptyset}\;[\textsc{TeVar}]$$

$$\frac{\Omega, \Gamma + \{x : \mu_1\} \vdash e : \mu_2, \varphi \quad \mu = (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \quad \Omega \vdash \mu \quad G(\Omega, \Gamma, e, \{x\}, \mu)}{\Omega, \Gamma \vdash \lambda x.e\ \mathtt{at}\ \rho : \mu, \{\rho\}}\;[\textsc{TeLam}]$$

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

$$\frac{\Omega, \Gamma \vdash e_1 : \mu_1, \varphi_1 \quad \Omega, \Gamma \vdash e_2 : \mu_2, \varphi_2}{\Omega, \Gamma \vdash \mathtt{(}e_1, e_2\mathtt{)}\ \mathtt{at}\ \rho : (\mu_1 \times \mu_2, \rho), \varphi_1 \cup \varphi_2 \cup \{\rho\}}\;[\textsc{TePair}] \qquad \frac{i \in \{1, 2\} \quad \Omega, \Gamma \vdash e : (\mu_1 \times \mu_2, \rho), \varphi}{\Omega, \Gamma \vdash \mathtt{\#}i\ e : \mu_i, \varphi \cup \{\rho\}}\;[\textsc{TeSel}]$$

$$\frac{\Omega, \Gamma \vdash e : \mu, \varphi \quad \{\rho, \vec{\epsilon}\} \cap \mathrm{frev}(\Omega, \Gamma, \mu) = \emptyset}{\Omega, \Gamma \vdash \mathtt{letregion}\ \rho\ \mathtt{in}\ e : \mu, \varphi \setminus \{\rho, \vec{\epsilon}\}}\;[\textsc{TeReg}] \qquad \frac{\Omega, \Gamma \vdash e_1 : \pi, \varphi_1 \quad \Omega, \Gamma + \{x : \pi\} \vdash e_2 : \mu, \varphi_2}{\Omega, \Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \mu, \varphi_1 \cup \varphi_2}\;[\textsc{TeLet}]$$

$$\frac{\begin{array}{c}\Omega + \Delta, \Gamma + \{x : \mu_1\} \vdash e : \mu_2, \varphi \quad \Omega \vdash \pi \\ (\mathrm{dom}(\Delta) \cup \mathrm{frev}(\vec{\rho}\vec{\epsilon})) \cap \mathrm{fv}(\Omega, \Gamma, \rho) = \emptyset \quad \pi = (\forall \vec{\rho}\vec{\epsilon}\Delta.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \quad G(\Omega, \Gamma, e, \{f, x\}, \pi)\end{array}}{\Omega, \Gamma \vdash \mathtt{fun}\ f\ [\vec{\rho}]\ x = e\ \mathtt{at}\ \rho : \pi, \{\rho\}}\;[\textsc{TeFun}]$$

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

Figure 4. Typing rules for values and expressions.

Proposition 12 (Typing Closed Under Type Substitution)**.** *If* $\Omega + \Delta, \Gamma \vdash e : \pi, \varphi$ *and* $\Omega \vdash S : \Delta$ *then* $\Omega, S(\Gamma) \vdash S(e) : S(\pi), S(\varphi)$.

Proof. By induction on the derivation of $\Omega + \Delta, \Gamma \vdash e : \pi, \varphi$. A detailed proof appears in Appendix A. □

Proposition 13 (Environment Extensibility)**.** *If* $\Omega, \Gamma \vdash e : \pi, \varphi$ *and* $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$ *then* $\Omega, \Gamma + \Gamma' \vdash e : \pi, \varphi$.

Proof. By induction on the derivation of $\Omega, \Gamma \vdash e : \pi, \varphi$. Notice in particular that the gc-safety relation is closed under environment extensibility. □

Proposition 14 (Type-Variable Context Extensibility)**.** *If* $\Omega, \Gamma \vdash e : \pi, \varphi$ *and* $\mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega') = \emptyset$ *then* $\Omega + \Omega', \Gamma \vdash e : \pi, \varphi$.

Proof. By induction on the derivation of $\Omega, \Gamma \vdash e : \pi, \varphi$. Notice in particular that the gc-safety relation is closed under type-variable context extensibility. □

Typed values contain no free program variables and the free variables of typed expressions are captured by the environment:

Proposition 15 (Free Variables)**.** *If* $\vdash v : \pi$ *then* $\mathrm{fpv}(v) = \emptyset$. *Moreover, if* $\Omega, \Gamma \vdash e : \pi, \varphi$ *then* $\mathrm{fpv}(e) \subseteq \mathrm{dom}(\Gamma)$.

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

Figure 5. The grammars for *evaluation contexts* (E) and *instructions* (ι).

Proof. By simultaneous induction on the derivations of $\vdash v : \pi$ and $\Omega, \Gamma \vdash e : \pi, \varphi$. □

Proposition 16 (Value Substitution). *If* $\Omega, \Gamma + \{x : \pi\} \vdash e : \pi', \varphi$ *and* $\vdash v : \pi$ *then* $\Omega, \Gamma \vdash e[v/x] : \pi', \varphi$.

Proof. By induction on the derivation of $\Omega, \Gamma + \{x : \pi\} \vdash e : \pi', \varphi$. For the cases that involve the gc-safety relation, Proposition 15 and Proposition 10 are applied. See Appendix A for details. □

3.10 A Small Step Dynamic Semantics

The dynamic semantics that we present is in the style of a contextual dynamic semantics [33] and is similar to the semantics given by Helsen and Thiemann [10, 26], although it differs in the way that inaccessibility to values in deallocated regions is 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 shown in Figure 5. 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 Figure 6 and consist of *allocation and deallocation rules*, *reduction rules*, and a *context rule*. The rules are of the form $e \overset{\varphi}{\longmapsto} e'$, 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, $e \overset{\varphi}{\longmapsto} e_1 \overset{\varphi}{\longmapsto} e_2 \overset{\varphi}{\longmapsto} \cdots$.

3.11 Type Safety

The proof of type safety is based on well-known techniques for proving type safety for statically typed languages [33, 50].

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

$$\lambda x.e\ \mathtt{at}\ \rho \overset{\varphi \cup \{\rho\}}{\longmapsto} \langle \lambda x.e \rangle^\rho \quad [\textsc{Lam}]$$

$$(v_1, v_2)\ \mathtt{at}\ \rho \overset{\varphi \cup \{\rho\}}{\longmapsto} \langle v_1, v_2 \rangle^\rho \quad [\textsc{Pair}]$$

$$\mathtt{fun}\ f\ \mathtt{[}\vec{\rho}\mathtt{]}\ x = e\ \mathtt{at}\ \rho \overset{\varphi \cup \{\rho\}}{\longmapsto} \langle \mathtt{fun}\ f\ \mathtt{[}\vec{\rho}\mathtt{]}\ x = e \rangle^\rho \quad [\textsc{Fun}]$$

$$\mathtt{letregion}\ \rho\ \mathtt{in}\ v \overset{\varphi}{\longmapsto} v \quad [\textsc{Reg}]$$

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

$$\langle \lambda x.e \rangle^\rho\ v \overset{\varphi \cup \{\rho\}}{\longmapsto} e[v/x] \quad [\textsc{App}]$$

$$\mathtt{let}\ x = v\ \mathtt{in}\ e \overset{\varphi}{\longmapsto} e[v/x] \quad [\textsc{Let}]$$

$$\begin{array}{l} \langle \mathtt{fun}\ f\ \mathtt{[}\vec{\rho}\mathtt{]}\ x = e \rangle^\rho\ \mathtt{[}\vec{\rho}\,'\mathtt{]}\ \mathtt{at}\ \rho' \overset{\varphi \cup \{\rho\}}{\longmapsto} \\ \quad \lambda x.e[\vec{\rho}\,'/\vec{\rho}][(\langle \mathtt{fun}\ f\ \mathtt{[}\vec{\rho}\mathtt{]}\ x = e \rangle^\rho)/f]\ \mathtt{at}\ \rho' \end{array} \quad [\textsc{Rapp}]$$

$$\#1\ \langle v_1, v_2 \rangle^\rho \overset{\varphi \cup \{\rho\}}{\longmapsto} v_1 \quad [\textsc{Sel1}]$$

$$\#2\ \langle v_1, v_2 \rangle^\rho \overset{\varphi \cup \{\rho\}}{\longmapsto} v_2 \quad [\textsc{Sel2}]$$

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

Figure 6. Evaluation rules.

We shall not present the complete proofs here, but refer the reader to [13], which includes proofs for a similar system.

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 17 (Unique Decomposition). *If* $\vdash e : \pi, \varphi$, *then either (1)* e *is a value, or (2) there exist 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, as well as progress and type soundness, can be stated as follows:

Proposition 18 (Type Preservation). *If* $\vdash e : \pi, \varphi$ *and* $e \overset{\varphi}{\longmapsto} e'$ *then* $\vdash e' : \pi, \varphi$.

Proof. By induction on the derivation $e \overset{\varphi}{\longmapsto} e'$. Details are provided in Appendix A. □

Proposition 19 (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 17 there exist 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 $\iota \overset{\varphi \cup \varphi'}{\longmapsto} e_2$, for some

e_2, so that $E_{\varphi'}[\iota] \overset{\varphi}{\longmapsto} E_{\varphi'}[e_2]$ follows from [Ctx] in Figure 6. Details are provided in Appendix A. □

Theorem 1 (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 18 and Proposition 19. □

We now introduce the notion of *context containment*, written $\varphi \models_{\mathrm{c}} e$, which expresses that when e can be written in the form $E_{\varphi'}[e']$, values in e' must be contained in the 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 Figure 7.

The following containment theorem states that, for well-typed programs, containment is preserved under evaluation:

Theorem 2 (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. □

Essentially, 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 time during evaluation, live reachable values are stored in regions that are either global or present on the region stack. This last property is essential for reference-tracing garbage collection, which relies on the safety of dereferencing live reachable values [34]. In particular, the containment theorem allows for a reference-tracing garbage collector to be interleaved with evaluation (as captured by the small-step evaluation semantics).

4 Implementation

For practical purposes, it is desirable to identify a quantified type variable to be *spurious* if it either appears free in the type of identifiers occurring free in a function expression (but not in the type of the function) or occurs free in a type that is instantiated for another spurious type variable. In particular, it turns out that only spurious type variables need to be associated with arrow effects in type variable contexts, which, in general, leads to simpler region type schemes, while limiting the computational overhead of applying effect substitutions. In the following we shall refer to a *spurious function* as one with spurious type variables in its inferred type scheme. It turns out that spurious functions occur only rarely in real programs. For example, the MLKit implementation of the entire Standard ML Basis Library [20] contains only three spurious functions, which include the top-level composition function `o` and the functions `Option.compose` and `Option.mapPartial`.

The region type system presented in the previous section extends to other ML-language features, including references, algebraic datatypes, and exceptions.

4.1 Region Inference

Region inference takes as input a well-typed source program and returns a region annotated version of the program that is well-typed according to the region typing rules. A simple region inference algorithm stores all values in the global region ρ and associates all function arrows and quantified occurrences of spurious type variables with the arrow effect $\epsilon.\{\rho\}$, where ϵ is a global effect variable. It is straightforward to prove that this trivial region inference algorithm leads to well-typed region-annotated programs and works for all source programs that are well-typed according to a classic Hindley-Milner style type system.

A proper region-inference algorithm introduces regions locally and seek to quantify over region variables and effect variables in order to pass regions to functions at runtime and to make it possible to use functions in different contexts without necessarily having to keep function arguments and results alive as long as the function is alive. In order to guarantee an upper limit to the number of introduced region variables and effect variables (to ensure termination), region inference can be divided into two phases. Here, the first phase, called the *spreading* phase, adds distinct fresh region variables to all allocation points and distinct fresh effect variables to all function type arrows. The second phase, called the *fix-point* phase, runs repeatedly until a fix-point is found by unifying region types according to the requirement of the region type system and by abstracting over region and effect variables, when possible, either by inserting `letregion` expressions or by abstracting over region variables and effect variables in `fun` expressions. The result is a well-typed region-annotated program. Implementing a proper region-inference algorithm for the region type system presented in the previous section differs from previous approaches by having to deal properly with spurious type variables and their associated arrow effects.

4.2 The MLKit

The MLKit is a Standard ML compiler that compiles programs to efficient native machine code for Linux and macOS [14] and implements a number of techniques for refining the representations of regions [6, 43], including dividing regions into stack allocated (bounded) regions (also called *finite regions*) and heap allocated regions (also called *infinite regions*), which are the regions that are subject to reference-tracing garbage collections.

The region type system presented in Section 3 is implemented in the MLKit in terms of a region-inference algorithm that deals properly with spurious type variables. The changes to the region inference algorithm are orthogonal to many of the later region-representation phases of the MLKit, including dropping of quantified parameter regions that are not stored into by a function and distinguishing between regions

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

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

Figure 7. Context containment.

holding different types of values (for supporting tag-free representations of values of certain types).

We emphasize here that the implementation changes, as proposed by the modified region-type system, are of mandatory importance for ensuring soundness of integrating region-inference and reference-tracing garbage collection.

The MLKit compiles all of Standard ML, including itself and the MLton compiler. MLton and the MLKit are two very different compilers with different characteristics. Whereas MLton generates very compact (and often very efficient) executables, by featuring aggressive inlining and optimisation strategies, the MLKit features efficient recompilation and relative fast compilation for large programs. For instance, compiling the MLKit from scratch with MLKit itself takes 201 seconds (real time) whereas the same task takes 1039 seconds with MLton.[4] Moreover, upon changes of source code, recompiling the the MLKit with the MLKit compiler often takes less than 10 seconds.

In some cases, a spurious function can be rewritten as a non-spurious function. Consider first the function `List.app` from the Standard ML Basis Library. This function has type scheme $\forall\alpha.(\alpha \to \texttt{unit}) \to \alpha\ \texttt{list} \to \texttt{unit}$ with the following possible implementation:

```
fun app f =
  let fun loop nil = ()
        | loop (x::xs) = (f x ; loop xs)
  in loop
  end
```

Unfortunately, a Standard ML compiler based on algorithm W [32] will give `app` the type scheme $\forall\alpha\beta.(\alpha \to \beta) \to \alpha\ \texttt{list} \to \texttt{unit}$ and `loop` the type $\alpha\ \texttt{list} \to \texttt{unit}$ (a module signature constraint may later constrain the type scheme of `app` to be less generic). Because `f` has type $\alpha \to \beta$ and occurs free in `loop`, β is inferred to be a spurious type variable. In general, the number of inferred spurious type variables may be decreased by applying a type minimization algorithm [7]. For the example with `app`, it suffices to give a direct type constraint to the function, specifying that `f` has type $\alpha \to \texttt{unit}$.

As a second example, the Standard ML Basis Library contains a function `Array.copy`, which copies elements from a generic source array (of type $\alpha\ \texttt{array}$) into locations in a target array (also of type $\alpha\ \texttt{array}$). One possible implementation of this function uses a local utility function `loop` that, assuming no overlaps, loops through the indexes of the source and at each index, fetches the corresponding value from the source array and updates the appropriate location in the target array. This local function will have type $\texttt{int} \to \texttt{unit}$, which means that the type variable α will be inferred to be spurious. In practice, the inference of α to be spurious will likely have little influence on region-inference for programs that use the `Array.copy` function. It is possible, however, also in this case, to modify the code slightly, by passing the source array as an additional argument to the loop function, in order to ensure that α is not considered spurious.

4.3 Tracking Spurious Type-Variable Dependencies

It may be enlightening to see how the type system tracks spurious type variable dependencies. Consider the program in Figure 8(a). This program is much similar to the program presented in the introduction, except that the spurious type variable bound by the composition function `o` is here not instantiated to a ground type immediately. Instead, it is instantiated to a new spurious type variable, which is bound by the function `g` with the type scheme $\forall\alpha.(\texttt{unit} \to \alpha) \to \texttt{unit} \to \texttt{unit}$.

There are a couple of interesting aspects about the region-annotated version of the program, which appears in Figure 8(b). First, notice how the inference algorithm has arranged for the two intermediate functions (passed to the composition function `o`) to be stored in the same region ρ_5, which is bound by (and passed to) the function `g`. This unification is due to the region inference algorithm unifying secondary quantified region and effect variables in type schemes, which is a central part of ensuring termination of region inference. Second, consider the region type scheme for the function `g`:

$$\begin{aligned}&\forall\rho_5\rho_6\rho_7\epsilon_1\epsilon_2\epsilon\epsilon_4(\alpha : \epsilon.\emptyset).\\ &\quad(\texttt{unit} \xrightarrow{\epsilon_1.\emptyset} \alpha, \rho_7) \quad \xrightarrow{\epsilon_2.\{\epsilon_1,\rho_7,\rho_5,\rho_6\}}\\ &\quad(\texttt{unit} \xrightarrow{\epsilon_4.\{\epsilon,\rho_5\}} \texttt{unit}, \rho_6)\end{aligned}$$

We see that α is inferred to be a spurious type variable and that it is associated with the arrow effect $\epsilon.\emptyset$. The reason α

[4] All measurements are performed on a MacBook Pro (15-inch, 2016), 16GB RAM, 2.7 GHz Quad-Core Intel Core i7, running MLKit v4.7.2 and MLton 20220831.211529-gd6080abba with option `-disable-pass deepFlatten`.

```
fun g (f : unit->'a)
  : unit->unit =
  op o
  let val x = f()
  in (fn x => (),
      fn () => x)
  end
val h =
  g (fn () => "oh" ^ "no")
```

(a)

```
fun g [ρ5, ρ6] f =
  letregion ρ3
  in op o[ρ6] let val x = f ()
                    in (fn at ρ5 x => (),
                        fn at ρ5 v110 => x) at ρ3
                    end
  end
val h =
  letregion ρ4
  in g[ρ1,ρ1] (fn at ρ4 () => op ^[ρ2] ("oh", "no"))
  end
```

(b)

Figure 8. A problematic source code program featuring a dependency between two spurious type variables (a) and a sound region-annotated version of the program (b).

is inferred to be spurious is not because it appears free in the type of a variable captured in a closure but because it appears free in a type instantiated for another spurious type variable, namely that occurring in the type scheme for the function o. Notice that the type scheme for g captures that the argument function is applied immediately (ϵ_1 appears in the effect $\epsilon_2.\{\epsilon_1, \rho_7, \rho_5, \rho_6\}$. Here is the type instance of the type scheme, with ϵ'_1, ϵ'_2, ϵ', ϵ'_4 being fresh effect variables:

$$(\texttt{unit} \xrightarrow{\epsilon'_1.\emptyset} (\texttt{string}, \rho_2), \rho_4) \xrightarrow{\epsilon'_2.\{\epsilon'_1, \rho_4, \rho_1, \rho_2\}} (\texttt{unit} \xrightarrow{\epsilon'_4.\{\epsilon', \rho_1, \rho_2\}} \texttt{unit}, \rho_1)$$

We see that region ρ_4 does not occur in the type of the function returned by g, which perfectly aligns with the fact that the function passed to g, which is stored in ρ_4, is applied immediately and not accessed again. For this reason, region inference can surround the call to g with a `letregion` ρ_4 `in` ... end construct. Contrary, the region type scheme for g captures the relationship between the spurious quantified type variable α and the capture of a value of type α in the returned closure through the associated effect variable ϵ, which occurs in the effect of the resulting function. As a consequence, the string "ohno" is rightfully forced into a global region (i.e., ρ_2).

4.4 Type Variables in Exception Types

For the full Standard ML language, there is one other language feature that may lead to dangling pointers and that can also be controlled through the notion of spurious type variables. In Standard ML, local exception constructors may be declared with free type variables occuring in their argument types. For example, if a type variable 'a is bound explicitly by a function, a local exception declaration, occuring inside the body of the function, may take the form

```
exception E of 'a
```

Because a constructed exception value may escape to top-level (in case the exception value is raised), it is paramount that all regions holding the exception value (and perhaps its argument) are top-level regions.[5] By treating 'a as a spurious type variable and by associating it with a top-level effect variable, we are guaranteed that whenever the function with the local exception declatation is instantiated, all regions occuring in the type instantiated for 'a are forced to be top-level regions. Without threating 'a as a spurious type variable, it is straightforward to construct a program that will introduce a dangling pointer at runtime and cause the reference tracing garbage collector to fail.

5 Benchmarks

In this section, we report on the consequences of the type system changes for a variety of benchmark programs. We compare the benchmark programs using three different compilation strategies using the MLKit (v4.7.2) and a single compilation strategy using MLton (20220831.211529-gd6080abba) [49], a whole-program optimising Standard ML compiler, which serves to relate the performance of the code generated by the MLKit with the performance of a state-of-the-art compiler. We emphasise again that the MLKit and MLton are two very different compilers that, however, both generate native x64 machine code.

All benchmark programs are executed on a MacBook Pro (15-inch, 2016) with a 2.7GHz Intel Core i7 processor and 16GB of memory running macOS. Times reported are wall clock times and memory usage is measured using the macOS /usr/bin/time program. The benchmark programs span from micro-benchmarks such as fib37 and tak (7 and

[5] The MLKit implementation does not attempt to infer when or if raised exceptions are properly handled.

12 lines), which use only the runtime stack for allocation, to larger programs, such as vliw and mlyacc (3681 and 7385 lines), that solve real-world problems.

The three MLKit compilation strategies include the **rg** compilation strategy, which is based on the region type-system presented in this paper and which combines region-inference and reference-tracing garbage collection, the **rg-** compilation strategy, which is like **rg** but without taking spurious type variables into account (and which is therefore unsound), and, finally, the **r** compilation strategy, which is based alone on region-inference.

Figure 9 lists the benchmark programs and reports on how the type system changes influence the generated code for each of the benchmarks. Measurements are averages over 10 runs. The **real time** columns list the average execution time in seconds, annotated with relative standard deviations. For the **rss** and **gc #** columns, the relative standard deviations are less than 1 percent. There are a number of observations to made. First notice that, for many of the programs, the type system has no influence on the generated code (and thus on region live ranges) even in cases where many of the functions are spurious and when boxed types are instantiated for spurious type variables (column **diff**). We also see that for programs that contain no spurious functions (column **fcns**), the type system changes have no influence on the generated code (column **diff**). However, for certain programs containing spurious functions, even when there are no instantiations of boxed types for spurious type variables (column **inst**), the type system changes may have resulted in different generated code in terms of longer region live ranges (programs barnes-hut, kbc, simple, zebra, and zern). There are two reasons why generated code may be different in these cases. The first reason may be that the implementation identifies the effect variable associated with a spurious type variable with the effect variable associated with the function type for which the type variable appears free in the type of a free variable, as illustrated by (3). The second reason may be that the implementation unifies secondary effect variables, which may lead to unifying of effects that are otherwise unrelated.

Concerning execution times (the **real time** columns), we see that there are no significant differences between the execution times for the **rg** and **rg-** strategies, even for cases where the generated code differs (due to different region live ranges). Notice also that for none of the benchmarks do we experience failures due to the possibility of dangling-pointers in the **rg-** compilation strategy. We also see that the **r** compilation strategy performs better than the **rg** and **rg-** strategies. Sometimes MLKit generates faster code than MLton, which is the case for DLX, fib37, mlyacc, msort, simple, and tsp), but, for most benchmarks, MLton outperforms the MLKit.

With respect to memory usage (the **rss** columns), we see that the **rg** and **rg-** compilation strategies have similar behavior. We also see that the **r** compilation strategy sometimes perform better (e.g., fft), which is due to its more compact (tag-free) value representation and the less-restrictive region type system (dangling pointers are permitted). Sometimes, however, reference-tracing garbage collection is essential, which is exemplified by the benchmarks barnes-hut, logic, nucleic, and zebra. We also see that the memory usage of MLton generated executables is often higher than the memory usage of the **rg** compilation strategy (we have not explored MLKit's and MLton's runtime flags for adjusting heap-to-live ranges, etc.)

Finally, from the **gc #** columns, we see that, across the benchmarks, the **rg** and **rg-** compilation strategies lead to executables that trigger similar numbers of garbage collections (we cannot explain the difference for the zebra benchmark.)

6 Related Work

Most related to this work is the previous work on combining region inference and garbage collection in the MLKit [24], the work on integrating region-based memory management and generational garbage collection [16], and the previous work on guaranteeing the absence of dangling pointers for region-based memory management [13]. Compared to previous work, the present work does not aim at distinguishing between regions containing different types of values, but is concerned purely about establishing a sound foundation for integrating region inference and reference-tracing garbage collection. The region type system (and the region inference algorithm) presented in this paper integrates well with the techniques for typing regions. These techniques allow for a tag-free representation of pairs, triples, and references, which provides dramatic savings on allocated memory and execution time.

Another strand of related work is the large body of related work concerning general garbage collection techniques [28] and garbage collection techniques for functional languages, including [11, 27, 36, 48]. Incremental, concurrent, and real-time garbage collection techniques for functional languages have recently obtained much attention. In particular, the presence of generations has been shown useful for collecting parts of the heap incrementally and in a concurrent and parallel fashion [4, 30, 31]. We leave it to future work to investigate the use of regions and generations in the MLKit for supporting concurrency and parallelism in the language.

There is also a series of proposals for tag-free garbage collection schemes [1, 5, 22, 23, 47] and nearly tag-free garbage collection schemes [35, 40]. The partly tag-free garbage collection scheme supported by the region type system 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, as mentioned, can

					real time (s)				**rss (Mb ± 1.2%)**				**gc #**	
Program	**loc**	**fcns**	**inst**	**diff**	**rg**	**rg-**	**r**	MLton	**rg**	**rg-**	**r**	MLton	**rg**	**rg-**
DLX	2841	2/149	0/690	✓	0.15 ± 6%	0.15 ± 5%	*0.14 ± 10%*	0.41 ± 5%	7	7	7	32	3	3
barnes-hut	1245	2/140	0/459	✓	0.60 ± 2%	0.64 ± 3%	0.58 ± 2%	*0.15 ± 7%*	4	4	170	2	473	473
fft	73	0/19	0/45		0.56 ± 1%	0.58 ± 3%	0.45 ± 3%	*0.25 ± 8%*	69	69	56	128	11	11
fib37	7	0/1	0/0		0.41 ± 5%	0.41 ± 7%	*0.27 ± 3%*	0.36 ± 4%	2	2	2	1	1	1
kbc	679	1/90	0/249	✓	0.24 ± 7%	0.24 ± 7%	0.21 ± 3%	*0.10 ± 8%*	11	9	9	2	11	10
lexgen	1322	0/108	0/531		0.71 ± 8%	0.68 ± 4%	0.57 ± 2%	*0.40 ± 4%*	14	14	67	18	109	109
life	202	0/35	0/146		0.53 ± 5%	0.54 ± 6%	0.45 ± 3%	*0.44 ± 4%*	2	2	14	2	58	58
logic	351	0/22	0/806		0.49 ± 2%	0.50 ± 4%	0.34 ± 2%	*0.13 ± 10%*	3	3	255	2	1844	1844
mandelbrot	62	0/5	0/0		0.38 ± 5%	0.37 ± 3%	0.37 ± 5%	*0.32 ± 6%*	2	2	2	1	1	1
mlyacc	7385	10/966	5/3256	✓	0.32 ± 4%	*0.31 ± 3%*	0.32 ± 3%	0.34 ± 2%	17	14	116	11	29	28
mpuz	124	0/13	0/44		0.76 ± 6%	0.73 ± 2%	0.49 ± 3%	*0.29 ± 3%*	2	2	2	1	2	2
msort-rf	119	0/14	0/27		0.63 ± 4%	0.64 ± 3%	*0.51 ± 4%*	1.00 ± 6%	118	118	98	654	16	16
msort	113	0/13	0/22		0.89 ± 4%	0.87 ± 2%	*0.57 ± 5%*	1.02 ± 4%	132	132	381	388	26	26
nucleic	3215	1/40	475/1273		0.30 ± 2%	0.31 ± 9%	0.33 ± 3%	*0.20 ± 10%*	4	4	235	2	645	645
professor	282	0/57	0/99		0.45 ± 3%	0.45 ± 5%	0.38 ± 2%	*0.33 ± 7%*	3	3	10	1	263	263
ratio	620	0/54	0/848		1.39 ± 2%	1.38 ± 2%	1.27 ± 6%	*0.38 ± 3%*	16	16	36	47	14	14
ray	529	1/48	0/120		0.69 ± 3%	0.72 ± 5%	0.62 ± 1%	*0.25 ± 1%*	13	13	13	14	12	12
simple	1053	15/327	0/448	✓	0.28 ± 3%	0.25 ± 8%	*0.15 ± 3%*	0.28 ± 7%	4	4	3	7	4	4
tak	12	0/2	0/0		0.80 ± 2%	0.84 ± 6%	0.81 ± 2%	*0.63 ± 9%*	2	2	2	1	1	1
tsp	493	0/26	0/19		0.13 ± 3%	0.14 ± 9%	*0.12 ± 8%*	0.16 ± 3%	10	10	5	11	7	7
vliw	3681	5/563	4/2133	✓	0.60 ± 4%	0.60 ± 3%	0.50 ± 5%	*0.31 ± 4%*	13	14	44	9	15	15
zebra	313	2/50	0/288	✓	1.35 ± 2%	1.35 ± 3%	1.29 ± 3%	*0.45 ± 4%*	3	3	123	1	336	404
zern	605	3/103	0/34	✓	0.66 ± 3%	0.71 ± 5%	0.46 ± 5%	*0.34 ± 5%*	4	4	4	11	4503	4503

Figure 9. Benchmark programs. The second column (**loc**) lists the size of the benchmark in terms of lines of code, excluding Basis Library code. The third column (**fcns**) lists the number of spurious functions, relative to the total number of functions. The fourth column (**inst**) lists the number of times a spurious type variable is instantiated with a boxed type, relative to the total number of type variable instantiations. The fifth column (**diff**) indicates if the notion of spurious type variables made a difference to the generated target program. The next four columns (**real time**) list execution times in seconds for different benchmark compilation strategies. The next three columns (**rss**) list memory usage (in Mb) for the compilation strategies. Finally, the last two columns list the number of reference tracing garbage collections for the strategies **rg** and **rg-**.

lead to significant time and memory savings, in particular because pairs and triples are used for the implementation of many dynamic data structures, including lists and trees.[6]

As other techniques that support full untagging, our technique does not involve traversing the runtime stack to determine types during garbage collection [5, 22, 23] or require special type information to be passed to functions at runtime [47]. 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 [25].

Another body of related work investigates the notion of escape analysis for improving stack allocation in garbage collected systems [8, 38]. Region inference and MLKit's polymorphic multiplicity analysis [6] allow more objects to be stack allocated than traditional escape analyses, which allows only local, non-escaping values to be stack allocated. Other work investigates the use of static prediction techniques and linear typing for inferring heap space usage [29].

Cyclone [39] is a region-based type-safe C dialect, for which, the programmer can decide if an object should reside in the GC heap or in a region. Cyclone is constructed to disallow program code to dereference dangling pointers. For the GC heap, Cyclone uses a conservative reference-tracing collector and no guarantee is given that it does not trace dangling pointers (safety is ensured by the collector being conservative). Another region-based language is Gay and Aiken's RC system, which features limited explicit regions

[6] The scheme works well together with support for unboxed data constructors, such as cons (: :), which, for instance, leads to a compact representation of linked lists [12].

for C, combined with reference counting of regions [21]. A modern language for system programming is Rust, which is based on ownership types for controlling the use of resources, including memory [3]. Ownership types are also used for real-time implementations of Java [9]. None of the above systems are combined with techniques for reference-tracing garbage collection of each individual region (Cyclone allows values to be stored in the global garbage collected heap region, but other regions are not collected using reference-tracing collection). Ownership types also lead to problems with constructing cyclic data structures, which are straightforward to work with in effect-based systems.

Also related to the present work is the work by Aiken et al. [2], who show how region inference may be improved for some programs by removing the constraints of the stack discipline, which may cause a garbage collector to run less often. Other work in this area includes [19], which removes the constraints of the region stack discipline for an intermediate language using a linear type system.

Region inference has also been used in practical settings without combining it with reference-tracing garbage collection. In particular, it has been used as the primary memory management scheme for a web server [15, 18].

7 Conclusion and Future Work

We have identified and fixed a soundness problem with combining region inference and reference-tracing garbage collection. The solution involves associating so-called spurious type variables with effect sets and tracking effect dependencies to ensure that no dangling pointers appear during evaluation of a program. The work thus justifies earlier work by (1) suggesting how the unsafe type system is modified into a sound type system and (2) demonstrating that the necessary modifications to the region type system have little influence on the generated code and thus, on previous reported results on combining region inference and reference-tracing garbage collection.

There are multiple paths of relevant future work. Whereas the type system presented in this paper has been proven sound on paper, we do not have a mechanised version of the proof, which would be a major engineering task. We consider efforts in this direction as possible future work. Another possibility for future work is on allowing programmers to interfere with region inference by being explicit about regions and effects in types and expressions. Finally, a possibility for future work would be to improve instruction selection and optimisations of MLKit programs to match the performance of MLton executables in more cases.

From a sustainability point-of-view, region inference seems like a viable technique for limiting the memory footprint of programs as garbage collections can occur less frequently if a part of the heap is managed by explicit memory allocation and deallocation. Future work may investigate this path in more details.

References

[1] Shail Aditya, Christine H. Flood, and James E. Hicks. 1994. Garbage Collection for Strongly-Typed Languages Using Run-Time Type Reconstruction. In *LISP and Functional Programming*. 12–23. citeseer.nj.nec.com/32465.html

[2] Alexander Aiken, Manuel Fähndrich, and Raph Levien. 1995. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. In *ACM Conference on Programming Languages and Implementation (PLDI '95)*.

[3] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. 2002. Alias Annotations for Program Understanding. In *ACM Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*.

[4] Todd A. Anderson. 2010. Optimizations in a Private Nursery-based Garbage Collector. In *ACM International Symposium on Memory Management (ISMM '10)*.

[5] Andrew W. Appel. 1989. Runtime tags aren't necessary. *Lisp and Symbolic Computation* 2 (1989), 153–162.

[6] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. 1996. From Region Inference to von Neumann Machines via Region Representation Inference. In *ACM Symposium on Principles of Programming Languages (POPL '96)*.

[7] Nikolaj Skallerud Bjørner. 1994. Minimal Typing Derivations. In *ACM SIGPLAN Workshop on ML and its Applications*. 120–126.

[8] Bruno Blanchet. 1998. Escape Analysis : Correctness Proof, Implementation and Experimental Results. In *ACM Symposium on Principles of Programming Languages (POPL'98)*. ACM Press, 25–37.

[9] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. 2003. Ownership Types for Safe Region-based Memory Management in Real-time Java. In *ACM Conference on Programming Language Design and Implementation (PLDI '03)*.

[10] Cristiano Calcagno, Simon Helsen, and Peter Thiemann. 2002. Syntactic Type Soundness Results for the Region Calculus. *Information and Computation* 173, 2 (2002).

[11] Damien Doligez and Xavier Leroy. 1993. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *ACM Symposium on Principles of Programming Languages (POPL '93)*.

[12] Martin Elsman. 1998. Polymorphic Equality—No Tags Required. In *Second International Workshop on Types in Compilation*.

[13] Martin Elsman. 2003. Garbage Collection Safety for Region-based Memory Management. In *ACM Workshop on Types in Language Design and Implementation (TLDI '03)*.

[14] Martin Elsman and Niels Hallenberg. 1995. An Optimizing Backend for the ML Kit Using a Stack of Regions. Student Project 95-7-8, University of Copenhagen (DIKU).

[15] Martin Elsman and Niels Hallenberg. 2003. Web Programming with SMLserver. In *International Symposium on Practical Aspects of Declarative Languages (PADL'03)*. Springer-Verlag.

[16] Martin Elsman and Niels Hallenberg. 2020. On the Effects of Integrating Region-Based Memory Management and Generational Garbage Collection in ML. In *Practical Aspects of Declarative Languages (PADL '20)*. Springer International Publishing, 95–112.

[17] Martin Elsman and Niels Hallenberg. 2021. Integrating region memory management and tag-free generational garbage collection. *Journal of Functional Programming* 31 (2021), e4. https://doi.org/10.1017/S0956796821000010

[18] Martin Elsman, Philip Munksgaard, and Ken Friis Larsen. 2018. Experience Report: Type-Safe Multi-Tier Programming with Standard ML Modules. In *Proceedings of the ML Family Workshop* (St. Louis, Missouri, USA) *(ML '18)*.

[19] Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear Regions Are All You Need. In *Programming Languages and Systems (ESOP '06)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 7–21.

[20] Emden R. Gansner and John H. Reppy. 2004. *The Standard ML Basis Library.* Cambridge University Press. https://doi.org/10.1017/CBO9780511546846

[21] David Gay and Alexander Aiken. 2001. Language Support for Regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01).* ACM Press, Snowbird, Utah.

[22] Benjamin Goldberg. 1991. Tag-free garbage collection for strongly typed programming languages. In *ACM Conference on Programming Language Design and Implementation.* 165–176.

[23] Benjamin Goldberg and Michael Gloger. 1992. Polymorphic Type Reconstruction for Garbage Collection Without Tags. In *LISP and Functional Programming.* 53–65. citeseer.nj.nec.com/goldberg92polymorphic.html

[24] Niels Hallenberg, Martin Elsman, and Mads Tofte. 2002. Combining Region Inference and Garbage Collection. In *ACM Conference on Programming Language Design and Implementation (PLDI'02).* ACM Press. Berlin, Germany.

[25] David R. Hanson. 1980. A portable storage management system for the Icon programming language. *Software—Practice and Experience* 10 (1980), 489–500.

[26] Simon Helsen and Peter Thiemann. 2000. Syntactic Type Soundness for the Region Calculus. In *International Workshop on Higher Order Operational Techniques in Semantics.* Published in Volume 41(3) of the Electronic Notes in Theoretical Computer Science..

[27] Lorenz Huelsbergen and Phil Winterbottom. 1998. Very Concurrent Mark-&-sweep Garbage Collection Without Fine-grain Synchronization. In *ACM International Symposium on Memory Management (ISMM '98).*

[28] Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall/CRC.

[29] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-order Programs. In *ACM Symposium on Principles of Programming Languages (POPL '10).*

[30] Simon Marlow and Simon Peyton Jones. 2011. Multicore Garbage Collection with Local Heaps. In *ACM International Symposium on Memory Management (ISMM '11).*

[31] Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2009. Runtime Support for Multicore Haskell. In *ACM International Conference on Functional Programming (ICFP '09).*

[32] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17 (1978), 348–375.

[33] Greg Morrisett. 1995. *Compiling with Types.* Ph. D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

[34] Greg Morrisett, Matthias Felleisen, and Robert Harper. 1995. Abstract Models of Memory Management. In *International Conference on Functional Programming Languages and Computer Architecture.* 66–77. San Diego.

[35] Greg Morrisett, David Tarditi, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. 1996. The TIL/ML Compiler: Performance and Safety through Types. citeseer.nj.nec.com/morrisett96tilml.html

[36] John H. Reppy. 1994. *A High-performance Garbage Collector for Standard ML.* Technical Report. AT&T Bell Laboratories.

[37] Guillaume Salagnac, Chaker Nakhli, Christophe Rippert, and Sergio Yovine. 2006. Efficient Region-Based Memory Management for Resource-limited Real-Time Embedded Systems.. In *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems.*

[38] G. Salagnac, S. Yovine, and D. Garbervetsky. 2005. Fast Escape Analysis for Region-based Memory Management. *Electron. Notes Th. C. S.* 131 (May 2005), 99–110.

[39] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006. Safe Manual Memory Management in Cyclone. *Sci. Comput. Program.* 62, 2 (Oct. 2006), 122–144.

[40] David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. 1996. TIL: A Type-Directed Optimizing Compiler for ML. In *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation.* 181–192. citeseer.nj.nec.com/tarditi95til.html

[41] Mads Tofte and Lars Birkedal. 1998. A Region Inference Algorithm. *Transactions on Programming Languages and Systems (TOPLAS)* 20, 4 (July 1998), 734–767.

[42] Mads Tofte and Lars Birkedal. 2000. Unification and Polymorphism in Region Inference. *Proof, Language, and Interaction. Essays in Honour of Robin Milner* (May 2000). (25 pages).

[43] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation* 17, 3 (01 Sep 2004), 245–265.

[44] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. 2021. *Programming with Regions in the MLKit (Revised for Version 4.6.0).* Technical Report. Department of Computer Science, University of Copenhagen, Denmark.

[45] Mads Tofte and Jean-Pierre Talpin. 1993. *A Theory of Stack Allocation in Polymorphically Typed Languages.* Technical Report DIKU-report 93/15. Department of Computer Science, University of Copenhagen.

[46] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997), 109–176.

[47] Andrew P. Tolmach. 1994. Tag-Free Garbage Collection Using Explicit Type Parameters. In *LISP and Functional Programming.* 1–11. citeseer.nj.nec.com/52227.html

[48] Katsuhiro Ueno and Atsushi Ohori. 2016. A Fully Concurrent Garbage Collector for Functional Programs on Multicore Processors. In *ACM International Conference on Functional Programming (ICFP '16).*

[49] Stephen Weeks. 2006. Whole-Program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML* (Portland, Oregon, USA) *(ML '06).* Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/1159876.1159877

[50] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. citeseer.nj.nec.com/wright92syntactic.html

A Detailed proofs

The below detailed proofs follow closely the structure of the proofs provided in [13], but adjusted to treat type variable contexts properly.

Proposition 12 (Typing Closed Under Type Substitution). *If* $\Omega + \Delta, \Gamma \vdash e : \pi, \varphi$ *and* $\Omega \vdash S : \Delta$ *then* $\Omega, S(\Gamma) \vdash S(e) : S(\pi), S(\varphi)$.

Proof. By induction on the derivation of $\Omega + \Delta, \Gamma \vdash e : \pi, \varphi$. The cases for integers (values), pairs (values and expressions), projections (expressions), and identifiers are trivial.

Case $e = \langle \lambda x.e' \rangle^{\rho}$. From [TvLam], we have $\Omega + \Delta, \Gamma \vdash e : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \emptyset$ and $\{\}, \{x : \mu_1\} \vdash e' : \mu_2, \varphi$ and $\vdash \pi$. Because $\mathrm{ftv}(\pi, e', \varphi) \cap \mathrm{dom}(S) = \emptyset$, we have $\{\}, \{x : S(\mu_1)\} \vdash S(e') : S(\mu_2), S(\varphi)$ and $\vdash S(\pi)$. Moreover, because $\mathrm{frv}(\pi) = \mathrm{frv}(S(\pi))$, we have $\mathrm{frv}(S(\pi)) \models_{\mathrm{v}} S(e')$. We can now apply [TvLam] to get $\Omega, S(\Gamma) \vdash S(e) : S(\pi), \emptyset$, as required.

Case $e = e_1\ e_2$. From [TeApp], we have $\Omega + \Delta, \Gamma \vdash e : \mu, \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}$ and $\Omega + \Delta, \Gamma \vdash e_1 : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, \rho), \varphi_1$ and $\Omega + \Delta, \Gamma \vdash e_2 : \mu', \varphi_2$. By induction we have $\Omega, S(\Gamma) \vdash S(e_1) : (S(\mu') \xrightarrow{S(\epsilon.\varphi_0)} S(\mu), S(\rho)), S(\varphi_1)$ and $\Omega, S(\Gamma) \vdash S(e_2) : S(\mu'), S(\varphi_2)$. It follows from the definition of type substitution that $S(\epsilon.\varphi_0) = \epsilon.\varphi_0$. We can now apply [TeApp] to get $\Omega, S(\Gamma) \vdash S(e) : S(\mu), S(\varphi)$, as required.

Case $e = \texttt{letregion}\ \rho\ \texttt{in}\ e'$. From [TeReg], we have $\Omega + \Delta, \Gamma \vdash e : \mu, \varphi \setminus \{\rho, \vec{\epsilon}\}$ and $\Omega + \Delta, \Gamma \vdash e' : \mu, \varphi$ and $\{\rho, \vec{\epsilon}\} \cap \mathrm{frev}(\Gamma, \mu) = \emptyset$. By renaming of bound names and because ρ and $\vec{\epsilon}$ do not appear free in the concluding type judgment, we can assume $\{\rho, \vec{\epsilon}\} \cap \mathrm{frev}(S(\Gamma), S(\mu)) = \emptyset$. By induction, we have $\Omega, S(\Gamma) \vdash S(e') : S(\mu), S(\varphi)$. We can now apply [TeReg], to get $\Omega, S(\Gamma) \vdash \texttt{letregion}\ \rho\ \texttt{in}\ S(e') : S(\mu), S(\varphi) \setminus \{\rho, \vec{\epsilon}\}$. It follows trivially that we have $\Omega, S(\Gamma) \vdash S(e) : S(\mu), S(\varphi \setminus \{\rho, \vec{\epsilon}\})$, as required.

Case $e = e'\ [\vec{\rho}]\ \texttt{at}\ \rho$. From [TeRApp], we have $\Omega + \Delta, \Gamma \vdash e' : (\sigma, \rho'), \varphi$ and $\Omega + \Delta \vdash \sigma \geq \tau$ via $\vec{\rho}$ and $\Omega \vdash \tau$. By induction, we have $\Omega, S(\Gamma) \vdash S(e') : (S(\sigma), S(\rho')), S(\varphi)$. From Proposition 7, we have $\Omega \vdash S(\sigma) \geq S(\tau)$ via $\vec{\rho}$, thus, from [TeRApp], we can conclude $\Omega, S(\Gamma) \vdash S(e) : S(\tau, \rho), S(\varphi \cup \{\rho, \rho'\})$, as required.

Case Rule [TeSub]. We have $\Omega + \Delta, \Gamma \vdash e : \pi, \varphi$ and $\Omega + \Delta, \Gamma \vdash e : \pi, \varphi'$ and $\varphi' \supseteq \varphi$. By induction. we have $\Omega, S(\Gamma) \vdash S(e) : S(\mu), S(\varphi)$. From the definition of type substitution, it follows that $\varphi' \supseteq \varphi$ implies $S(\varphi') \supseteq S(\varphi)$, thus, we can apply [TeSub] to get $\Omega, S(\Gamma) \vdash S(e) : S(\mu), S(\varphi')$, as required. □

Proposition 16 (Value Substitution). *If* $\Omega, \Gamma + \{x : \pi\} \vdash e : \pi', \varphi$ *and* $\vdash v : \pi$ *then* $\Omega, \Gamma \vdash e[v/x] : \pi', \varphi$.

Proof. By induction on the derivation $\Omega, \Gamma + \{x : \pi\} \vdash e : \pi', \varphi$.

Case $e = y$. From assumptions and [TeVar], we have $\Omega, \Gamma + \{x : \pi\} \vdash y : \pi', \varphi$ and $(\Gamma + \{x : \pi\})(y) = \pi'$ and $\varphi = \emptyset$. If $y \neq x$, we have $e[v/x] = y$, thus, because $\Gamma(y) = \pi'$, we can conclude from [TeVar] that $\Omega, \Gamma \vdash e[v/x] : \pi', \varphi$, as required. Otherwise, $y = x$, thus $e[v/x] = v$ and $\pi = \pi'$. From assumptions, [TeVal], and [TeSub], we have $\Omega, \Gamma \vdash e[v/x] : \pi', \varphi$, as required.

Case $e = \lambda y.e'\ \texttt{at}\ \rho$. From assumptions and [TeLam], we have $\Omega, \Gamma + \{x : \pi, y : \mu\} \vdash e' : \mu', \varphi'$ and $\varphi = \{\rho\}$ and $\pi' = (\mu \xrightarrow{\epsilon.\varphi'} \mu', \rho)$. By renaming of bound variables, we can assume $x \neq y$, thus, we can apply the induction hypothesis to get $\Omega, \Gamma + \{y : \mu\} \vdash e'[v/x] : \mu', \varphi'$. By applying [TeLam], we have $\Omega, \Gamma \vdash \lambda y.e'[v/x] : \pi', \varphi$, as required.

The remaining cases follow similarly. □

Proposition 17 (Unique Decomposition). *If* $\vdash e : \pi, \varphi$, *then either (1)* e *is a value, or (2) there exist 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. Suppose e is not a value. There are 8 cases to consider. We proceed by case analysis.

Case $e = \texttt{letregion}\ \rho\ \texttt{in}\ e_1$. A derivation $\vdash e : \pi, \varphi$ must end in a use of [TeReg] followed by a number of uses of [TeSub]. It follows that there exist φ_1 and φ_2 such that $\varphi = \varphi_1 \setminus \{\rho\} \cup \varphi_2$ and $\rho \notin \mathrm{frv}(\pi)$ and $\vdash e_1 : \pi, \varphi_1$. By renaming of bound variables, we can assume $\rho \notin \mathrm{frv}(\varphi_2)$. By induction, either e_1 is a value or there exist a unique $E'_{\varphi''}$, ι_1, and π'_1 such that $e_1 = E'_{\varphi''}[\iota_1]$ and $\vdash \iota_1 : \pi'_1, \varphi_1 \cup \varphi''$. If e_1 is not a value then we take $E_{\varphi'} = \texttt{letregion}\ \rho\ \texttt{in}\ E'_{\varphi''}$, $\varphi' = \varphi'' \cup \{\rho\}$, $\iota = \iota_1$, $\pi' = \pi'_1$, and from [TeSub], we have $\vdash \iota_1 : \pi'_1, \varphi \cup \varphi'$, because $\varphi_1 \cup \varphi'' \subseteq \varphi \cup \varphi'$. Otherwise, $e_1 = v_1$ for some value v_1. Thus, $E_{\varphi'} = [\cdot]$, $\iota = \texttt{letregion}\ \rho\ \texttt{in}\ v_1$, $\pi' = \pi$, and $\varphi' = \emptyset$.

Case $e = e_1\ e_2$. A derivation $\vdash e : \pi, \varphi$ must end in a use of [TeApp], followed by a number of uses of [TeSub]. It follows that there exist μ, φ_1, φ_2, μ', ϵ, φ_0, and φ_3 such that $\varphi = \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\} \cup \varphi_3$ and $\vdash e_1 : (\mu \xrightarrow{\epsilon.\varphi_0} \mu', \rho), \varphi_1$ and $\vdash e_2 : \mu, \varphi_2$ and $\pi = \mu'$. By induction, either e_1 is a value or else there exist $E'_{\varphi'_1}$, ι_1, and π'_1 such that $e_1 = E'_{\varphi'_1}[\iota_1]$ and $\vdash \iota_1 : \pi'_1, \varphi_1 \cup \varphi'_1$. If e_1 is not a value, then we take $E_{\varphi'} = E'_{\varphi'_1}\ e_2$, $\iota = \iota_1$, $\pi' = \pi'_1$, and because $\varphi' = \varphi'_1$ and $\varphi_1 \subseteq \varphi$, we can apply [TeSub] to get $\vdash \iota_1 : \pi'_1, \varphi \cup \varphi'$. Otherwise, $e_1 = v_1$ for some value v_1. We can now apply the induction hypothesis to get that either e_2 is a value or else there exist $E'_{\varphi'_2}$, ι_2, and π'_2 such that $e_2 = E'_{\varphi'_2}[\iota_2]$ and $\vdash \iota_2 : \pi'_2, \varphi_2 \cup \varphi'_2$. If e_2 is not a value, then we take $E_{\varphi'} = v_1\ E'_{\varphi'_2}$, $\iota = \iota_2$, $\pi' = \pi'_2$, and because $\varphi' = \varphi'_2$ and $\varphi_2 \subseteq \varphi$, we can apply [TeSub] to get $\vdash \iota_2 : \pi'_2, \varphi \cup \varphi'$. Otherwise $e_2 = v_2$ for some value v_2. Because $\vdash v_1 : (\mu \xrightarrow{\epsilon.\varphi_0} \mu', \rho), \varphi_1$, we can conclude from inspecting the typing rules for values (canonical forms) that $v_1 = \langle \lambda x.e' \rangle^{\rho}$. Thus, $E_{\varphi'} = [\cdot]$, $\varphi' = \emptyset$, $\iota = \langle \lambda x.e' \rangle^{\rho}\ v_2$, and $\pi' = \pi$.

The remaining 6 cases follow similarly. □

Proposition 18 (Type Preservation). *If* $\vdash e : \pi, \varphi$ *and* $e \overset{\varphi}{\longmapsto} e'$ *then* $\vdash e' : \pi, \varphi$.

Proof. By induction on the structure of e. We proceed by case analysis.

Case $e = \lambda x.e_0\ \mathtt{at}\ \rho$. From assumptions and [TeLam], we have $\pi = (\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2, \rho)$ and $\{x : \mu_1\} \vdash e_0 : \mu_2, \varphi_0$ and $\vdash \pi$ and $G(\{\}, \{\}, e_0, \{x\}, \pi)$ and $\varphi = \{\rho\}$. From [Lam], we have $\rho \in \varphi$ and $e' = \langle \lambda x.e_0 \rangle^{\rho}$. From definition (4), we have $\mathrm{frv}(\pi) \models_{\mathrm{v}} e_0$. Now, by use of [TvLam] and [TeSub], we have $\vdash e' : \pi, \varphi$, as required.

Case $e = (v_1, v_2)\ \mathtt{at}\ \rho$. As above.

Case $e = \mathtt{fun}\ f\ [\vec{\rho}\,]\ x = e\ \mathtt{at}\ \rho$. As above.

Case $e = \mathtt{letregion}\ \rho\ \mathtt{in}\ v$. From assumptions and from [TeReg], there exist φ' and μ such that $\varphi = \varphi' \setminus \{\rho\}$ and $\vdash v : \mu, \varphi'$ and $\pi = \mu$. It follows from [TeVal] that $\vdash v : \mu, \emptyset$, thus, from [Reg] and [TeSub], we have $\vdash e' : \pi, \varphi$, as required.

Case $e = \langle \lambda x.e_1 \rangle^{\rho}\ v$. From assumptions, [TeApp], and [TvLam], there exist μ, μ_1, ϵ, and φ_0 such that $\pi = \mu$ and $\{x : \mu_1\} \vdash e_1 : \mu, \varphi_0$ and $\vdash v : \mu_1, \varphi_1$, and $\varphi = \varphi_0 \cup \{\epsilon, \rho\}$. From [TeVal], we have $\vdash v : \mu_1, \emptyset$. Thus, from Proposition 16, we have $\vdash e_1[v/x] : \mu, \varphi_0$. Now, because $\varphi \supseteq \varphi_0$, we can apply [TeSub] to get $\vdash e' : \pi, \varphi$, as required.

Case $e = \mathtt{let}\ x = v\ \mathtt{in}\ e'$. As above.

Case $e = \langle \mathtt{fun}\ f\ [\vec{\rho}\,]\ x = e_1 \rangle^{\rho}\ [\vec{\rho}\,']\ \mathtt{at}\ \rho'$. There are two possibilities. Either [TvFun] applies or [TvRec] applies.

case Rule [TvFun]. From assumptions, [TeRApp], and [TvFun], we have $\pi = (\tau, \rho')$, $\varphi = \{\rho, \rho'\}$, $v = \langle \mathtt{fun}\ f\ [\vec{\rho}\,]\ x = e_1 \rangle^{\rho}$ and $\sigma = \forall \vec{\rho}\vec{\epsilon}\Delta.\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2$, and

$$\vdash v : (\sigma, \rho) \tag{11}$$

$$\vdash \sigma \geq \tau \text{ via } \vec{\rho}\,' \tag{12}$$

$$\{\}, \{x : \mu_1\} \vdash e_1 : \mu_2, \varphi_0 \tag{13}$$

From (13), we have $f \notin \mathrm{fpv}(e_1)$, thus, we have

$$\{\}, \{x : \mu_1\} \vdash e_1[v/f] : \mu_2, \varphi_0 \tag{14}$$

From the definition of instantiation and from (12), there exists a substitution $S = (S^{\mathrm{t}}, [\vec{\rho}\,'/\vec{\rho}\,], S^{\mathrm{e}})$ such that

$$S(\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2) = \tau \tag{15}$$

$$\{\} \vdash S^{\mathrm{t}} : \Delta \tag{16}$$

From (14) and [TeLam], we have

$$\vdash \lambda x.e_1[v/f]\ \mathtt{at}\ \rho' : (\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2, \rho'), \{\rho'\} \tag{17}$$

By renaming of bound names, we can assume $S(v) = v$ and $S(\rho') = \rho'$, thus, from (15), (16), (17), Proposition 11, and Proposition 12, we have $\vdash \lambda x.e_1[\vec{\rho}\,'/\vec{\rho}\,][v/f]\ \mathtt{at}\ \rho' : (\tau, \rho'), \{\rho'\}$. We can now apply [TeSub] to get $\vdash e' : \pi, \varphi$, as required.

case Rule [TvRec]. From assumptions, [TeRApp], and [TvRec], we have $\pi = (\tau, \rho')$, $\varphi = \{\rho, \rho'\}$, $v = \langle \mathtt{fun}\ f\ [\vec{\rho}\,]\ x = e_1 \rangle^{\rho}$ and $\sigma = \forall \vec{\rho}\vec{\epsilon}\Delta.\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2$, and

$$\vdash v : (\sigma, \rho) \tag{18}$$

$$\sigma' = \forall \vec{\rho}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2 \tag{19}$$

$$\vdash \sigma \geq \tau \text{ via } \vec{\rho}\,' \tag{20}$$

$$\{f : (\sigma', \rho)\}, x : \mu_1\} \vdash e_1 : \mu_2, \varphi_0 \tag{21}$$

From (18), (19), and [TvRec], we have

$$\vdash v : (\sigma', \rho) \tag{22}$$

From Proposition 16 and (22) and (21), we have

$$\{x : \mu_1\} \vdash e_1[v/f] : \mu_2, \varphi_0 \tag{23}$$

From the definition of instantiation and from (20), there exists a substitution $S = (S^{\mathrm{t}}, [\vec{\rho}\,'/\vec{\rho}\,], S^{\mathrm{e}})$ such that

$$S(\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2) = \tau \tag{24}$$

$$\{\} \vdash S^{\mathrm{t}} : \Delta \tag{25}$$

From (23) and [TeLam], we have

$$\vdash \lambda x.e_1[v/f]\ \mathtt{at}\ \rho' : (\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2, \rho'), \{\rho'\} \tag{26}$$

By renaming of bound names, we can assume $S(v) = v$ and $S(\rho') = \rho'$, thus, from (24), (25), (26), Proposition 11, and Proposition 12, we have $\vdash \lambda x.e_1[\vec{\rho}\,'/\vec{\rho}\,][v/f]\ \mathtt{at}\ \rho' : (\tau, \rho'), \{\rho'\}$. We can now apply [TeSub] to get $\vdash e' : \pi, \varphi$, as required.

Case $e = \mathtt{\#1}\ (v_1, v_2)$. From assumptions, [TeSel], and [TvPair], we have $\vdash v_1 : \mu, \emptyset$. We can now apply [TeSub] to get $\vdash v_1 : \mu, \varphi$, as required.

Case $e = E_{\varphi'}[e'']$. We have $e'' \overset{\varphi \cup \varphi'}{\longmapsto} e'''$ and $\varphi \cap \varphi' = \emptyset$ and $e' = E_{\varphi'}[e''']$. We now proceed by case analysis on the structure of $E_{\varphi'}$.

case $E_{\varphi'}[e''] = (e'', e_2)\ \mathtt{at}\ \rho$. We have $\varphi' = \emptyset$. From assumptions and [TePair] we have $\vdash e'' : \mu_1, \varphi_1$, $\vdash e_2 : \mu_2, \varphi_2$, $\mu = (\mu_1 \times \mu_2, \rho)$, and $\varphi = \varphi_1 \cup \varphi_2 \cup \{\rho\}$. By applying [TeSub], we have $\vdash e'' : \mu_1, \varphi$. We can now apply the induction hypothesis to get $\vdash e''' : \mu_1, \varphi$. By applying [TePair], we have $\vdash E_{\varphi'}[e'''] : \mu, \varphi$, as required.

case $E_{\varphi'}[e''] = (v_1, e'')\ \mathtt{at}\ \rho$. We have $\varphi' = \emptyset$. From assumptions and [TePair] we have $\vdash v_1 : \mu_1, \varphi_1$, $\vdash e'' : \mu_2, \varphi_2$, $\mu = (\mu_1 \times \mu_2, \rho)$, and $\varphi = \varphi_1 \cup \varphi_2 \cup \{\rho\}$. By applying [TeSub], we have $\vdash e'' : \mu_2, \varphi$. We can now apply the induction hypothesis to get $\vdash e''' : \mu_2, \varphi$. By applying [TePair], we have $\vdash E_{\varphi'}[e'''] : \mu, \varphi$, as required.

case $E_{\varphi'}[e''] = \mathtt{\#}i\ e''$, $i \in \{1, 2\}$. We have $\varphi' = \emptyset$. From assumptions and [TeSel], we have $\vdash e'' : (\mu_1 \times \mu_2, \rho), \varphi'$, $\mu = \mu_i$ and $\varphi = \varphi' \cup \{\rho\}$. By applying [TeSub], we have $\vdash e'' : (\mu_1 \times \mu_2, \rho), \varphi$, thus, we can apply the induction hypothesis to get $\vdash e''' : (\mu_1 \times \mu_2, \rho), \varphi$. We can now apply [TeSel] to get $\vdash E_{\varphi'}[e'''] : \mu, \varphi$, as required.

case $E_{\varphi'}[e''] = \mathtt{let}\ x = e''\ \mathtt{in}\ e_2$. We have $\varphi' = \emptyset$. From assumptions and [TeLet], there exists π such that $\vdash e'' : \pi, \varphi_1$, $\{x : \pi\} \vdash e_2 : \mu, \varphi_2$, and $\varphi = \varphi_1 \cup \varphi_2$. Applying [TeSub], we have $\vdash e'' : \pi, \varphi$. By induction, we have $\vdash e''' : \pi, \varphi$. We can now apply [TeLet] to get $\vdash E_{\varphi'}[e'''] : \mu, \varphi$, as required.

case $E_{\varphi'}[e''] = e''\ e_2$. From assumptions and [TeApp], it follows that there exist ϵ, φ_0, φ_1, φ_2, and ρ such that $\vdash e'' : (\mu_2 \xrightarrow{\epsilon.\varphi_0} \mu, \rho), \varphi_1$, $\vdash e_2 : \mu_2, \varphi_2$, and $\varphi = \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}$. From [TeSub], we have $\vdash e'' : (\mu_2 \xrightarrow{\epsilon.\varphi_0} \mu, \rho), \varphi$, thus, by induction, we have $\vdash e''' : (\mu_2 \xrightarrow{\epsilon.\varphi_0} \mu, \rho), \varphi$. We can now apply [TeApp] to get $\vdash E_{\varphi'}[e'''] : \mu, \varphi$, as required.

case $E_{\varphi'}[e''] = v\ e''$. As above.

case $E_{\varphi'}[e''] = e''\ [\vec{\rho}]\ \mathtt{at}\ \rho$. As above.

case $E_{\varphi'}[e''] = \mathtt{letregion}\ \rho\ \mathtt{in}\ e''$. We have $\varphi' = \{\rho\}$. From assumptions and from [TeReg], there exist φ'' and $\vec{\epsilon}$ such that $\varphi = \varphi'' \setminus \{\rho, \vec{\epsilon}\}$, and $\vdash e'' : \mu, \varphi''$. From [TeSub], we have $\vdash e'' : \mu, \varphi \cup \varphi'$. We can now apply the induction hypothesis to get $\vdash e''' : \mu, \varphi \cup \varphi'$. Now, because $\varphi = (\varphi \cup \varphi') \setminus \{\rho, \vec{\epsilon}\}$, we can apply [TeReg] to get $\vdash E_{\varphi'}[e'''] : \mu, \varphi$, as required.

The remaining cases follow similarly. □

Proposition 19. (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 17 there exist a unique $E_{\varphi'}$, ι, and π' such that $e = E_{\varphi'}[\iota]$ and $\vdash \iota : \pi', \varphi \cup \varphi'$. We argue that $\iota \overset{\varphi\cup\varphi'}{\longmapsto} e_2$, for some e_2, so that $E_{\varphi'}[\iota] \overset{\varphi}{\longmapsto} E_{\varphi'}[e_2]$ follows from [Ctx]. We now consider all cases where ι could possibly be stuck.

Case $\iota = \lambda x.e_1'\ \mathtt{at}\ \rho$. We have $\vdash \lambda x.e_1'\ \mathtt{at}\ \rho : \pi', \varphi \cup \varphi'$. This derivation must be an application of [TeLam] followed by a number of applications of [TeSub]. Thus, we have $\rho \in \varphi \cup \varphi'$. It follows that we can apply [Lam] to get $e_2 = \langle \lambda x.e_1' \rangle^{\rho}$.

Case $\iota = \langle \lambda x.e_{\mathrm{x}} \rangle^{\rho}\ v$. We have $\vdash \langle \lambda x.e_{\mathrm{x}} \rangle^{\rho}\ v : \pi', \varphi \cup \varphi'$. This derivation must end in an application of [TeApp] followed by a number of applications of [TeSub]. Thus, by applying [TeVal], there exist μ, μ', ϵ, and φ_0 such that $\vdash \langle \lambda x.e_{\mathrm{x}} \rangle^{\rho} : (\mu \xrightarrow{\epsilon.\varphi_0} \mu', \rho), \emptyset$ and $\vdash v : \mu, \emptyset$ and $\pi' = \mu'$ and $\varphi_0 \cup \{\epsilon, \rho\} \subseteq \varphi \cup \varphi'$. Now, because $\rho \in \varphi \cup \varphi'$, we can apply [App] to get $e_2 = e_{\mathrm{x}}[v/x]$.

Case $\iota = \langle \mathtt{fun}\ f\ [\vec{\rho}]\ x = e_0 \rangle^{\rho'}\ [\vec{\rho}\,']\ \mathtt{at}\ \rho$. The derivation $\vdash \iota : \pi', \varphi \cup \varphi'$ must end in an application of [TeRapp] followed by a number of applications of [TeSub], thus, from [TeVal], there exist σ and τ' such that $\pi' = (\tau', \rho)$ and

$$\vdash \langle \mathtt{fun}\ f\ [\vec{\rho}]\ x = e_0 \rangle^{\rho'} : (\sigma, \rho'), \emptyset \tag{27}$$

$$\{\rho, \rho'\} \subseteq \varphi \cup \varphi' \tag{28}$$

Because $\rho' \in \varphi \cup \varphi'$ follows from (28), we can apply [Rapp] to get $e_2 = \lambda x.e_0[\vec{\rho}\,'/\vec{\rho}][v/f]\ \mathtt{at}\ \rho$, where $v = \langle \mathtt{fun}\ f\ [\vec{\rho}]\ x = e_0 \rangle^{\rho'}$.

Case $\iota = \mathtt{letregion}\ \rho\ \mathtt{in}\ v$. It follows immediately from [Reg] that $e_2 = v$.

The remaining cases follow similarly. □