Explicit Effects and Effect Constraints in ReML

MARTIN ELSMAN, University of Copenhagen, Denmark

An important aspect of building robust systems that execute on dedicated hardware and perhaps in constrained environments is to control and manage the effects performed by program code.

We present ReML, a higher-order statically-typed functional language, which allows programmers to be explicit about the effects performed by program code and in particular effects related to memory management. Allowing programmers to be explicit about effects, the regions in which values reside, and the constraints under which code execute, makes programs robust to changes in the program source code and to compiler updates, including compiler optimisations.

ReML is integrated with a polymorphic inference system that builds on top of region-inference, as it is implemented in the MLKit, a Standard ML compiler that uses region-based memory management as its primary memory management scheme.

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

Additional Key Words and Phrases: Region-inference, Effect Systems, Parallelism, Memory Management

ACM Reference Format:
Martin Elsman. 2024. Explicit Effects and Effect Constraints in ReML. *Proc. ACM Program. Lang.* 8, POPL, Article 79 (January 2024), 25 pages. https://doi.org/10.1145/3632921

1 INTRODUCTION

Region inference is a memory discipline that aims at inserting instructions for allocating and deallocating memory in a program at compile time, without relying on collecting or maintaining liveness information dynamically as is usually the case for dynamic garbage collection techniques such as reference tracing garbage collection [Tofte and Birkedal 1998; Tofte and Talpin 1997].

To make region inference tractable, memory is organised into a stack of regions, each of which may grow dynamically (organised in pages). New regions are pushed on top of the region stack and the top-most region may be popped (thereby freeing the pages) when it is certain that no values residing in the top-most region will be needed for the remainder of the computation. Each allocating expression (e.g., pair construction or string concatenation) is annotated with a region variable and each expression e may be translated into an expression of the form `letregion` ρ `in` e, which follows the semantics that first a region (specified by ρ) is pushed (on the region stack), then the expression e is evaluated to a value v, perhaps using the region bound to ρ, and finally, the region is popped, with v being the result of evaluating the `letregion` construct. Functions may be inferred to take regions as arguments, which allows for modular development, meaning that a function may allocate in and read from regions that are not allocated when the function is defined.

Region inference builds on a region- and effect-based program analysis [Jouvelot and Gifford 1991; Lucassen and Gifford 1988; Talpin and Jouvelot 1994], which at its core tracks the memory effects (reads and writes) of expressions through an extended notion of types, where, in particular,

Author's address: Martin Elsman, Department of Computer Science, University of Copenhagen, Universitetsparken 5, Copenhagen, Denmark, DK-2100, mael@di.ku.dk.

ACM 2475-1421/2024/1-ART79
https://doi.org/10.1145/3632921

each function type is annotated with a so-called *latent effect*, a conservative approximation of the effect of evaluating functions of that type.

Region inference has been implemented for Standard ML in the MLKit compiler [Tofte et al. 2004, 2022] and augmented with reference-tracing garbage collection, which has been proven to be a viable alternative to purely dynamic garbage collection [Elsman 2023; Hallenberg et al. 2002].

Unfortunately, region inference alone falls short in some cases and it may be difficult for programmers to understand how to rewrite a program to become region friendly, let alone understand the memory requirements of a program. Although the MLKit features a region profiler that makes it possible to learn how regions are used and which allocations in the program contribute to potential excessive memory use [Hallenberg 1996; Tofte et al. 2022], properties of the program may change with minor changes to source code (or perhaps to the optimisation passes of the MLKit compiler).

We propose ReML, a Standard ML extension that allows programs to be annotated with information about in which region a value resides and the effects a function may have. A non-annotated Standard ML program is also a ReML program. The programmer may choose to annotate certain parts (and aspects) of a program while leaving other parts unannotated. Annotations may appear at allocation points or using type ascriptions. Regions and effects may be introduced through explicit `with` declarations or as additional function parameters. ReML will take the annotations seriously and interact with region inference to ensure that annotations are consistent with the underlying region-inference typing rules. In case of inconsistencies, errors are reported to the programmer. Moreover, ReML allows for specifying effect constraints through so-called `while`-types. Constraints include inclusion constraints (an effect contains a particular effect), disjunction constraints (two effects are non-overlapping), and basic constraints (e.g., an effect is empty).

We claim the following contributions:

(1) We present ReML, an ML language with incremental support for annotating programs with region and effects in a way that is faithful to an underlying region- and effect-based program analysis. We further devise a model for modular program development with effects that combines the notion of effect- and region-information in types and expressions with the notion of effect constraints.
(2) We develop a theoretical basis for effect constraints through a region- and effect-annotated language with explicit effect constraints and assertions. Based on a type system for the language and a small-step dynamic semantics, we demonstrate a soundness property for the language saying that well-typed programs do not get stuck.
(3) We describe how ReML is implemented on top of MLKit, a compiler for Standard ML, which is based on region- and effect-inference as the primary memory management discipline, and demonstrate how the underlying region-inference implementation cooperates with syntactic region- and effect-annotations in ReML.
(4) We give examples of practical programming in ReML and discuss extensions of ReML featuring exception effects, mutation effects, IO effects, and non-termination, which, together, form the basis for specifying pure-functions in ReML.

In the next section, we present an example of using effect constraints in ReML. The example aims at ensuring that a parallel version of Mergesort runs without allocation races. The example demonstrates the power of using effect constraints in conjunction with an effect inference system, leaving programmer annotations at a bare minimum, while allowing for modular development of high-performing code and, at the same time, providing a guarantee that an implementation will not suffer from allocation races. In Section 3, we present an explicitly region- and effect-annotated language with support for effect constraints and effect assertions and demonstrate a soundness result for the language. The language is essentially a region-annotated language based on the typed

```
fun pmsort (P:int) nil = nil
  | pmsort (P:int) xs =
 if P <= 1 then smsort xs
 else merge
       let val (l,r) = split xs
           val Q = P div 2
       in par (fn() ⇒ pmsort Q l,
               fn() ⇒ pmsort Q r)
       end
```

(a)

```
fun pmsort [ρ] (P:int) nil = nil
  | pmsort [ρ] (P:int) xs =
 if P <= 1 then smsort [ρ] xs
 else merge [ρ]
       let with ρ1 ρ2
           val (l,r) = split [ρ1,ρ2] xs
           val Q = P div 2
       in par (fn() ⇒ pmsort [ρ1] Q l,
               fn() ⇒ pmsort [ρ2] Q r)
       end
```

(b)

Fig. 1. Parallel Mergesort on Lists. Version (a) is without region annotations and version (b) is explicitly annotated with regions.

lambda-calculus augmented with support for recursive functions featuring type polymorphism and polymorphism in regions and effects and an assertion construct that halts evaluation unless an explicit effect constraint is satisfied. The language also features polymorphic recursion in regions and effects and a **letregion** construct for local bindings of regions and effects.

In Section 4, we present the syntactic constructs of ReML and show how the constructs interact with Standard ML syntax. We also give examples of how ReML complains if a program is not consistent with the region-inference typing rules.

In Section 5, we describe the notion of effect constraints using the notion of **while**-types. We also demonstrate the modular properties of working with effect constraints by showing how constraints from a calling context may be used for establishing the constraints of a called function. In particular, we continue the example concerning parallelism and demonstrate how the constraints of a low-level parallel primitive are satisfied by constraints specified by a higher-level construct (two functions may be evaluated in parallel without using allocation locks if the allocation effects of the two functions do not intersect).

In Section 6, we discuss aspects of ReML that allow the programmer to reason about other effects than memory effects, including exceptions, mutable updates, IO, non-termination, non-determination, and perhaps even user-defined effects. In Section 7, we describe how region and effects as well as effect constraints and constraint resolution are integrated with region inference. The section discusses the integration both from a theoretical perspective and from the perspective of the implementation in the MLKit compiler. Section 8 contains a discussion of a series of larger examples and gives an overview of the status of ReML and the artifact associated with the paper [Elsman 2024]. In Section 9, we describe related work and in Section 10, we conclude and discuss future work.

2 A MOTIVATING EXAMPLE FOR EFFECT CONSTRAINTS

As a first motivating example for effect constraints, consider the version of parallel Mergesort on lists shown in Figure 1(a).[1] The function `pmsort` takes as argument an integer `P` approximating the available parallel resources. When only one processor is available, `pmsort` reverts to a sequential version of Mergesort (provided by the function `smsort`). The function `pmsort` assumes a function

[1]We acknowledge that the parallel version of Mergesort shown here is not particularly parallel as a good parallel `merge` is difficult to implement with lists. A better Mergesort uses array slices and binary search for parallel merging.

`split:int list → int list * int list`, which takes a list and divides it into two approximately equally sized lists, and a function `merge:int list*int list → int list`, which, given two sorted lists, returns a sorted list containing the elements in the two argument lists. The strategy applied by `pmsort` is simple; if the argument is non-empty, split the argument into equally sized lists, sort the lists in parallel, and merge the sorted lists to produce the result. A function for executing code in parallel is the fork-join style operator `par`, which, in ReML, has the following type:

```
val par : (unit #e1→ α) * (unit #e2→ β) : α * β while e1 ## e2
```

The `par` function is specified to take a pair of two functions as argument and it returns a pair holding the result of evaluating the two functions.[2] Notice that each of the argument functions is annotated with a so-called effect variable and that the declaration is annotated with a so-called *effect constraint* `e1 ## e2`, which specifies that the two supplied functions should not have overlapping put-effects, meaning that they should not allocate into the same regions. In fact, it is critical for performance that the effect constraint is satisfied. Whereas the underlying runtime system will protect the allocation pointer in case of a race, a drastic performance penalty will appear if each thread makes many allocations into the shared region. We seek a lightweight method to ensure statically that there are no overlapping effects.

To appreciate properly the inferred region-annotated version of the code, which is listed in Figure 1(b), we first give the region- and effect-annotated types for `split` and `merge`, simplified slightly to serve the example:

$$\texttt{val split} : \forall\rho\rho_1\rho_2.(\texttt{int list},\rho) \xrightarrow{\{\texttt{get}(\rho),\texttt{put}(\rho_1),\texttt{put}(\rho_2)\}} (\texttt{int list},\rho_1) * (\texttt{int list},\rho_2)$$

$$\texttt{val merge} : \forall\rho_1\rho_2\rho.(\texttt{int list},\rho_1) * (\texttt{int list},\rho_2) \xrightarrow{\{\texttt{get}(\rho_1),\texttt{get}(\rho_2),\texttt{put}(\rho)\}} (\texttt{int list},\rho)$$

We see that `split` stores the resulting lists in two potentially different regions, possibly distinct from the region holding the argument list. Similarly, `merge` stores the merged list in a potentially different region from the regions holding the argument lists. Notice that in the region- and effect-annotated types for `split` and `merge`, $\texttt{put}(\rho)$ indicates the effect of storing into the region ρ and $\texttt{get}(\rho)$ indicates a read from the region ρ.

Looking now at the inferred region-annotated version of `pmsort` in Figure 1(b), we first see that `split` and `merge` are passed only the regions into which they allocate (for which there are `put` effects in their region-annotated types). We also see that support for region-polymorphic recursion has made it possible to use local regions (declared using an internal-language **`with`**-declaration) for storing the temporary sorted lists and that the regions into which the temporary lists are stored are disjoint as seen from the point-of-view of the `par` function. However, the inferred annotations provided in Figure 1(b) are fragile to changes in the source code. For instance, if the programmer provides a `merge` function that does not return its result in a region distinct from those holding the arguments, all threads will end up storing the merged lists in the same region. A traditional version of the function `merge` will have exactly this problem due to the typing rules for polymorphic type- and effect-inference:

```
fun merge (nil,ys) = ys
  | merge (xs,nil) = xs
  | merge (x::xs,y::ys) = if x < y then x :: merge(xs,y::ys)
                          else y :: merge(x::xs,ys)
```

[2]The `par` function is also the means to parallelism in MPL [Westrick et al. 2019], a Standard ML compiler, based on MLton, and equipped with fork-join style task-parallelism support.

Here is the region-annotated type of the above problematic version of `merge`:

```
val merge : ∀ρ.(int list,ρ) * (int list,ρ) --{get(ρ),put(ρ)}--> (int list,ρ)
```

ReML complaints with the following message if the constraint of the `par` function is violated:

```
** Error: par is passed two functions with intersecting put effects!
**  problematic effects: {put(r175)}
**  fun1: {put(r175)}
**  fun2: {put(r175)}
```

A fix is of course to perform a copy when one of the arguments is empty:

```
fun merge (nil,ys) = copy ys
  | merge (xs,nil) = copy xs
  | merge (x::xs,y::ys) = if x < y then x :: merge(xs,y::ys)
                          else y :: merge(x::xs,ys)
```

We shall see later, in Section 5.1, that `par` is not a primitive function in ReML but that the function is built on top of a more fundamental thread library and that the constraints specified by `par` are required to meet the specified constraints of the underlying functionality.

We emphasise here that effect constraints of the form discussed in this section is only part of what can be expressed with the technique described in this paper. Although the formalisation given in the next section focuses on providing disjointness guarantees about effect sets, the technique can be used also to reason about other effects, including exceptions, mutation, IO, and termination, as we shall discuss in Section 6.

3 FORMALISATION

We shall use ρ to range over so-called *region variables* and ϵ to range over so-called *effect variables*. An *effect* (φ) is a set of *atomic effects* (η), each of which can be either an effect variable or a region variable. Notice that, for the sake of the formalisation, we have simplified the notion of atomic effects and identified `get` and `put` effects. We shall later, in Section 6.1, return to the topic of refining the notion of atomic effects. Latent effects in the types of functions are of the form $\epsilon.\varphi$. Such objects are called *arrow effects* and are central for identifying effects and for defining the notion of substitution, which is the foundation for unification of region- and effect-annotated types and the region inference algorithm that we build upon [Tofte and Birkedal 1998, 2000].

A *basic constraint* (c) takes the form $\varphi \# \varphi$ and a *constraint set* (C) is a set of basic constraints. When $c = \varphi \# \varphi'$ is a basic constraint, we write $\mathrm{swap}(c)$ to denote the basic constraint $\varphi' \# \varphi$. We also write $\eta \# \eta'$ to mean $\{\eta\} \# \{\eta'\}$ when η and η' are atomic effects. A basic constraint c of the forms $\emptyset \# \varphi$ or $\varphi \# \emptyset$ is called a *nil-constraint* and we write $\mathrm{nil}(c)$ if c is a nil-constraint. A basic constraint $c = \varphi \# \varphi'$ is *valid*, written $\vdash c$ if $\varphi \cap \varphi' = \emptyset$. A constraint set C is *valid*, written $\vdash C$, if $\vdash c$ for all $c \in C$.

We shall use α to range over so-called *type variables*. Here are the definitions of atomic effects, effects, types, and type schemes, which are types parameterised over type variables, effect variables, and region variables:

τ	::=	α	– type variable	η	::=	$\epsilon \mid \rho$	– atomic effect
	$\mid$	`int`	– unboxed integer	φ	::=	$\{\eta_1, \cdots, \eta_n\}$	– effect
	$\mid$	$(\tau_1 \xrightarrow{\epsilon.\varphi} \tau_2, \rho)$	– boxed function	σ	::=	$\forall\vec{\alpha}.\sigma \mid \forall\vec{\epsilon}\vec{\rho}.\tau \triangleright C$	– type scheme

When τ is some type, we shall sometimes implicitly treat it as the type scheme with no quantified variables and the empty constraint set. Similarly, we shall often implicitly drop empty constraint sets in type schemes. When $\sigma = \forall\vec{\alpha}\vec{\epsilon}\vec{\rho}.\tau \triangleright C$, we consider $\vec{\alpha}$, $\vec{\epsilon}$, and $\vec{\rho}$ to be *bound* in τ and C and, in general, we consider objects identical up to renaming of bound names. Also, when o is some object, we write fv o to denote the set of free type variables, free region variables, and free effect variables in o. We also write frev o to denote the set of free region variables and free effect variables of o. Finally, we write fev o to denote the free effect variables of o.

3.1 Constraint Entailment

Constraint sets specify evidence that effects are disjoint. They are, in particular, used to reason about aliasing of region and effect parameters. Thus, when a function is defined, we may specify that two region parameters never alias. This property must then be established in each calling context. Moreover, region and effect variables introduced by **`letregion`**-constructs never alias other region and effect variables.

Whenever C is some constraint set, we write $C \# \varphi$ to specify the constraint $\varphi \# \varphi'$, where $\varphi' = \mathrm{frev}(C)$. Constraints of this form are established when φ denotes an effect set bound by a **`letregion`** construct, which, by definition, introduces regions and effects that are distinct from all other regions and effects in the context.

At (function) instantiation sites, we need to check that all instantiated constraints are satisfied, perhaps using constraints established in the calling context. For this purpose, we introduce the notion of *constraint entailment*. We first introduce a notion of *constraint normalisation*. When c is some basic constraint $\varphi \# \varphi'$, we write $||c||$ to denote the *normalised basic constraint set* $\{\eta \# \eta' \mid \eta \in \varphi, \eta' \in \varphi'\}$.

We say that a constraint set C *entails* another constraint set C', written $C \models C'$, if the judgment can be derived by the following rules:

Constraint Entailment $\boxed{C \models C'}$

$$\frac{C \cup ||c|| \models C'}{C \cup \{c\} \models C'}\ [\text{E-NormL}] \qquad \frac{\mathrm{frev}\, C \supseteq \mathrm{frev}\, c \quad C \models ||c|| \quad C \models C'}{C \models \{c\} \cup C'}\ [\text{E-NormR}] \qquad \frac{}{C \models \emptyset}\ [\text{E-Emp}]$$

$$\frac{c \in C \quad \vdash c \quad C \models C'}{C \models \{c\} \cup C'}\ [\text{E-Base}] \qquad \frac{\mathrm{swap}(c) \in C \quad \vdash c \quad C \models C'}{C \models \{c\} \cup C'}\ [\text{E-Swap}] \qquad \frac{\mathrm{frev}\, C \supseteq \mathrm{frev}\, c \quad \mathrm{nil}(c) \quad C \models C'}{C \models \{c\} \cup C'}\ [\text{E-Nil}]$$

Notice that if $C \models C'$ for some constraint sets C and C' then $C \cup C'' \models C'$, for any other constraint set C''. We refer to this property as *constraint entailment extensibility*.

An important property of constraint entailment is that validity is ensured for entailed constraint sets. That is, if $C \models C'$ then $\vdash C'$, which holds even if C is not valid. It is really the property of validity that is important for proving soundness (in particular progress) of the dynamic semantics. The machinery that we define here makes it possible to maintain validity of constraints also when functions are composed and instantiated in particular contexts.

3.2 Substitutions

A *substitution* (S) is a triple $(S^{\mathrm{r}}, S^{\mathrm{t}}, S^{\mathrm{e}})$, where S^{r} is a *region substitution*, mapping region variables to region variables, S^{t} is a *type substitution* mapping type variables to types, and S^{e} is an *effect substitution*, mapping effect variables to arrow effects. The effect of applying a substitution on an object is to perform the three substitutions simultaneously on the three kinds of variables in

the object (by renaming of bound variables within the object to avoid capture and by extending the maps to be the identity outside of their domains). For effects and arrow effects, substitution is defined as follows [Tofte and Birkedal 2000], assuming $S = (S^{\mathrm{r}}, S^{\mathrm{t}}, S^{\mathrm{e}})$:

$$S(\varphi) = \{S^{\mathrm{r}}(\rho) \mid \rho \in \varphi\} \cup \{\eta \mid \exists \epsilon, \epsilon', \varphi' \text{ s.t. } \epsilon \in \varphi \wedge S^{\mathrm{e}}(\epsilon) = \epsilon'.\varphi' \wedge \eta \in \{\epsilon'\} \cup \varphi'\}$$
$$S(\epsilon.\varphi) = \epsilon'.(\varphi' \cup S(\varphi)), \text{ where } S^{\mathrm{e}}(\epsilon) = \epsilon'.\varphi'$$

Substitutions form the basis for unification of region- and effect-annotated types in region inference and having substitutions map effect variables to arrow effects makes it possible to find unifiers (unifying substitutions) for two different region- and effect-annotated types with the same underlying ML types [Tofte and Birkedal 2000]. Whereas it may be possible to specify the region typing rules using another notion of substitution, we maintain consistency with region inference by using the same definition of substitution here. Substitutions may be composed (obeying function composition properties) and the restriction of a substitution S to some domain N is written $S \downarrow N$. Substitutions act as the identity outside of their domains.

In the following, we shall often work with substitutions that are particularly restricted. We therefore introduce the notation $C \models S \bullet C'$ to mean $C \models S(C')$ and $\mathrm{Dom}\, S \cap \mathrm{frev}\, C = \emptyset$.

The following proposition holds:

Proposition 3.1 (Entailment Substitution Closedness). *If $C \cup C' \models C''$ and $C \models S \bullet C'$ then $C \cup S(C') \models S(C'')$.*

Proof. By induction over the derivation of $C \cup C' \models C''$.

Case [E-Base]. We have $C'' = \{c\} \cup C'''$ and $C \cup C' \models C'''$ and $c \in (C \cup C')$ and $\vdash c$. Now, if $\mathrm{Dom}\, S \cap \mathrm{frev}\, c = \emptyset$, we have $S(c) \in (C \cup S(C'))$. If $\mathrm{Dom}\, S \cap \mathrm{frev}\, c \neq \emptyset$, we have $c \notin C$ because $\mathrm{Dom}\, S \cap \mathrm{frev}\, C = \emptyset$. Thus, $c \in C'$, from which it follows that $S(c) \in (C \cup S(C'))$ and because $\vdash S(C')$ follows from the assumptions, we have $\vdash S(c)$. By induction, we also have $C \cup S(C') \models S(C''')$. It follows from [E-Base] that $C \cup S(C') \models S(C''' \cup \{c\})$ and thus $C \cup S(C') \models S(C'')$, as required.

Case [E-NormL]. There are two cases. Either $c \in C$ or $c \notin C$.

If $c \in C$, then $\mathrm{Dom}\, S \cap \mathrm{frev}(c) = \emptyset$. It follows that $\mathrm{Dom}\, S \cap \mathrm{frev}(||c||) = \emptyset$. Let $C_0 = C \setminus \{c\}$. We have $(C_0 \cup ||c||) \cup C' \models C''$. From [E-NormL] and assumption, we have $C_0 \cup ||c|| \models S(C')$. We also have $\mathrm{Dom}\, S \cap \mathrm{frev}(C_0 \cup ||c||) = \emptyset$. Now, by induction, we have $(C \cup ||c||) \cup S(C') \models S(C'')$. By [E-NormL], we have $(C \cup \{c\}) \cup S(C') \models S(C'')$, and thus, $C \cup S(C') \models S(C'')$, as required.

If $c \notin C$, then $c \in C'$. Let $C'_0 = C' \setminus \{c\}$. It follows that $C' = C'_0 \cup \{c\}$. We thus have $C \cup (C'_0 \cup \{c\}) \models C''$ and $C \models S(C'_0 \cup \{c\})$. It follows that $(C \cup C'_0) \cup \{c\} \models C''$ and we can therefore apply [E-NormL] to get $C \cup (C'_0 \cup ||c||) \models C''$. From $C \models S(C'_0 \cup \{c\})$, we have $C \models S(C'_0) \cup \{S(c)\}$ and we can therefore apply [E-NormR] to get $C \models S(C'_0) \cup ||S(c)||$ and thus $C \models S(C'_0 \cup ||c||)$. Because we already have $\mathrm{Dom}\, S \cap \mathrm{frev}\, C = \emptyset$ from assumptions, we have by induction that $C \cup S(C'_0 \cup ||c||) \models S(C'')$ and thus $C \cup S(C'_0) \cup ||S(c)|| \models S(C'')$. From [E-NormL], we have $C \cup S(C'_0) \cup \{S(c)\} \models S(C'')$ and thus $C \cup S(C'_0 \cup \{c\}) \models S(C'')$. Because $C' = C'_0 \cup \{c\}$, we have $C \cup S(C') \models S(C'')$, as required.

The case for [E-NormR] follows similarly as the case for [E-NormL].

The cases for [E-Emp], [E-Nil], and [E-Swap] are straightforward. □

3.3 Instantiation

Given a constraint set C, we say that a type τ instantiates a type scheme $\sigma = \forall \vec{\alpha}\vec{\epsilon}\vec{\rho}.\tau' \triangleright C'$ via a substitution S, written $C \vdash \sigma \geq \tau$ via S, if the following conditions hold:

- $S(\tau') = \tau$
- $\mathrm{Dom}\, S = \{\vec{\alpha}\vec{\epsilon}\vec{\rho}\}$
- $C \models S(C')$

When we are interested in only the region instance list, we write $C \vdash \sigma \geq \tau$ via $\vec{\rho}'$ to mean, there exists a substitution $S = (S^{\mathrm{t}}, S^{\mathrm{r}}, S^{\mathrm{e}})$ such that $C \vdash \sigma \geq \tau$ via S and $\mathrm{Rng}\, S^{\mathrm{r}} = \vec{\rho}'$.

Notice that the relation requires that $S(C')$ can be proven based on the constraints in C. It is this requirement that ensures that the calling context is compatible with the function requirements.

Instantiation is closed under substitution according to the following proposition:

Proposition 3.2 (Instantiation Closed Under Substitution). *If* $C \cup C' \vdash \sigma \geq \tau$ via S' *and* $C \models S \bullet C'$ *then* $C \cup S(C') \vdash S(\sigma) \geq S(\tau)$ via S'', *where* $S'' = (S \circ S') \downarrow \mathrm{Dom}\, S'$.

Proof. Follows from the definition of instantiation. We have $\lceil 1 \rceil\ \sigma = \forall \vec{\alpha}\vec{\epsilon}\vec{\rho}.\tau' \triangleright C''$, $\lceil 2 \rceil\ S'(\tau') = \tau$, $\lceil 3 \rceil\ \mathrm{Dom}\, S' = \{\vec{\alpha}\vec{\epsilon}\vec{\rho}\}$, and $\lceil 4 \rceil\ C \cup C' \models S'(C'')$. From assumptions, $\lceil 4 \rceil$, and Proposition 3.1, we have $\lceil 5 \rceil\ C \cup S(C') \models S(S'(C''))$. By α-renaming, we have from $\lceil 2 \rceil$ and $\lceil 1 \rceil$ that $\lceil 6 \rceil\ S(\sigma) = \forall \vec{\alpha}\vec{\epsilon}\vec{\rho}.S(\tau') \triangleright S(C'')$ and $\lceil 7 \rceil\ S(S'(\tau')) = S(\tau)$. From assumptions, we have $S'' = (S \circ S') \downarrow \mathrm{Dom}\, S'$. Thus, from α-renaming and from $\lceil 3 \rceil$, we can assume $\mathrm{Dom}\, S' \cap \mathrm{fv}(S, C'') = \emptyset$. It follows that $\lceil 8 \rceil\ S(S'(\tau')) = S''(S(\tau'))$ and $\lceil 9 \rceil\ S(S'(C'') = S''(S(C''))$. From $\lceil 5 \rceil$, $\lceil 7 \rceil$, $\lceil 8 \rceil$, and $\lceil 9 \rceil$, we have $\lceil 10 \rceil\ C \cup S(C') \models S''(S(C''))$ and $\lceil 11 \rceil\ S''(S(\tau')) = S(\tau)$. We trivially have $\lceil 12 \rceil\ \mathrm{Dom}\, S'' = \{\vec{\alpha}\vec{\epsilon}\vec{\rho}\}$. Now, from the definition of instantiation and from $\lceil 6 \rceil$, $\lceil 11 \rceil$, $\lceil 12 \rceil$, and $\lceil 10 \rceil$, we have $C \cup S(C') \models S(\sigma) \geq S(\tau)$ via S'', as required. □

Also, based on extensibility of constraint entailment, it it straightforward to show that if $C \vdash \sigma \geq \tau$ via $\vec{\rho}$ then $C \cup C' \vdash \sigma \geq \tau$ via $\vec{\rho}$, for any C'.

3.4 Expressions

Expressions (e) and values (v) take the following forms:

e	::=	$x \mid v \mid \lambda x.e$ **at** $\rho \mid e\, e'$	– expressions	v	::=	$d \mid \langle \lambda x.e \rangle^{\rho}$	– value
	|	**letregion** φ **in** e	– local effect		|	$\langle$**fun** $f : \sigma\ [\vec{\rho}]\ x = e \rangle^{\rho}$	– function
	|	**assert** c **in** e	– assertion				
	|	**let** $x = e$ **in** e'	– binding				
	|	**fun** $f : \sigma\ [\vec{\rho}]\ x = e$ **at** ρ	– function				
	|	$e\ [\vec{\rho}]\ e'$	– application				

Values include unboxed integers (d), boxed ordinary closures, and boxed recursive function closures. Whereas a boxed value is annotated with information about in which region the value is located, the expression counterparts are annotated with information about in which region the value is stored when the allocating expression is evaluated. An expression can be a value, a variable, a lambda-expression, a let-expression, a function application, a recursive function expression, or a recursive function application. The language also includes an expression of the form **assert** c **in** e, which, when c takes the form $\{\rho\} \# \{\rho'\}$, has the effect of testing whether the two regions ρ and ρ' are indeed different before evaluation proceeds into e. If the two regions are identical, evaluation is stuck. Thus, one purpose of the type system is to guarantee that a well-typed expression is not stuck due to an assertion. We shall sometimes write **assert** $\rho \# \rho'$ **in** e to mean **assert** $\{\rho\} \# \{\rho'\}$ **in** e. The purpose of having the **assert** construct in the formal language is exactly to model operations such as parallel constructs that require certain constraints to be satisfied.

Notice that recursive function definitions may take regions as arguments and are annotated with explicit type schemes, which allow for specifying constraint set assertions. By allowing functions to be annotated with constraint set assertions, function preconditions can be established without a calling context knowing the internals of a function. At the same time, each function body can be type-checked in isolation from contexts that apply the function.

In expressions **let** $x = e'$ **in** e and $\lambda x.e$ **at** ρ, the variable x is *bound* in e. In expressions of the form **fun** $f : \forall \vec{\alpha}.\forall \vec{\epsilon}\vec{\rho}.\tau \triangleright C\ [\vec{\rho}]\ x = e$ **at** ρ, the variables f, $\vec{\alpha}$, $\vec{\epsilon}$, $\vec{\rho}$, and x are *bound* in e. Similarly

Typing Rules for Values $\boxed{C \vdash v : \sigma}$

$$\frac{}{C \vdash d : \mathtt{int}}\ [\text{TV-Int}] \qquad \frac{C, \{x : \tau\} \vdash e : \tau', \varphi'}{C \vdash \langle \lambda x.e \rangle^{\rho} : (\tau \xrightarrow{\epsilon.\varphi'} \tau', \rho)}\ [\text{TV-Lam}]$$

$$\frac{\begin{array}{c}\sigma = \forall \vec{\epsilon}\vec{\rho}.(\tau \xrightarrow{\epsilon.\varphi} \tau', \rho) \triangleright C' \qquad C'' = \{\eta \# \emptyset \mid \eta \in \{\vec{\epsilon}\vec{\rho}\}\} \\ \{\vec{\alpha}\vec{\epsilon}\vec{\rho}\} \cap \mathrm{fv}(C, \rho) = \emptyset \quad C \cup C' \cup C'', \{f : \sigma, x : \tau\} \vdash e : \tau', \varphi\end{array}}{C \vdash \langle \mathtt{fun}\ f : \forall \vec{\alpha}\sigma\ [\vec{\rho}]\ x = e \rangle^{\rho} : \forall \vec{\alpha}\sigma}\ [\text{TV-Fun}]$$

Typing Rules for Expressions $\boxed{C, \Gamma \vdash e : \sigma, \varphi}$

$$\frac{C \vdash v : \sigma}{C, \Gamma \vdash v : \sigma, \emptyset}\ [\text{T-Val}] \qquad \frac{\Gamma(x) = \sigma}{C, \Gamma \vdash x : \sigma, \emptyset}\ [\text{T-Var}] \qquad \frac{\varphi' \supseteq \varphi \quad C, \Gamma \vdash e : \sigma, \varphi}{C, \Gamma \vdash e : \sigma, \varphi'}\ [\text{T-Sub}]$$

$$\frac{C \cup \{\mathrm{frev}\ C \# \varphi\} \cup \{\eta_1 \# \eta_2 \mid \eta_1, \eta_2 \in \varphi, \eta_1 \neq \eta_2\}, \Gamma \vdash e : \tau, \varphi' \quad \varphi \cap \mathrm{frv}(C, \Gamma, \tau) = \emptyset}{C, \Gamma \vdash \mathtt{letregion}\ \varphi\ \mathtt{in}\ e : \tau, \varphi' \backslash \varphi}\ [\text{T-Reg}]$$

$$\frac{\begin{array}{c}C, \Gamma \vdash e : \sigma, \varphi \\ C, \Gamma + \{x : \sigma\} \vdash e' : \sigma', \varphi'\end{array}}{C, \Gamma \vdash \mathtt{let}\ x = e\ \mathtt{in}\ e' : \sigma', \varphi \cup \varphi'}\ [\text{T-Let}] \qquad \frac{\begin{array}{c}C, \Gamma \vdash e : (\tau \xrightarrow{\epsilon.\varphi_0} \tau', \rho), \varphi \\ C, \Gamma \vdash e' : \tau, \varphi'\end{array}}{C, \Gamma \vdash e\ e' : \tau', \{\rho, \epsilon\} \cup \varphi_0 \cup \varphi \cup \varphi'}\ [\text{T-App}]$$

$$\frac{C, \Gamma \vdash e : \tau, \varphi \quad C \models \{c\}}{C, \Gamma \vdash \mathtt{assert}\ c\ \mathtt{in}\ e : \tau, \varphi \cup \mathrm{frev}(c)}\ [\text{T-As}] \qquad \frac{C, \Gamma + \{x : \tau\} \vdash e : \tau', \varphi'}{C, \Gamma \vdash \lambda x.e\ \mathtt{at}\ \rho : (\tau \xrightarrow{\epsilon.\varphi'} \tau', \rho), \{\rho\}}\ [\text{T-Lam}]$$

$$\frac{\begin{array}{c}\sigma = \forall \vec{\epsilon}\vec{\rho}.(\tau \xrightarrow{\epsilon.\varphi} \tau', \rho) \triangleright C' \quad C'' = \{\eta \# \emptyset \mid \eta \in \{\vec{\epsilon}\vec{\rho}\}\} \\ \{\vec{\alpha}\vec{\epsilon}\vec{\rho}\} \cap \mathrm{fv}(C, \Gamma, \rho) = \emptyset \quad C \cup C' \cup C'', \Gamma + \{f : \sigma, x : \tau\} \vdash e : \tau', \varphi\end{array}}{C, \Gamma \vdash \mathtt{fun}\ f : \forall \vec{\alpha}\sigma\ [\vec{\rho}]\ x = e\ \mathtt{at}\ \rho : \forall \vec{\alpha}\sigma, \{\rho\}}\ [\text{T-Fun}]$$

$$\frac{C, \Gamma \vdash e : \sigma, \varphi \quad C, \Gamma \vdash e' : \tau, \varphi' \quad C \vdash \sigma \geq (\tau \xrightarrow{\epsilon.\varphi_0} \tau', \rho)\ \mathrm{via}\ \vec{\rho}}{C, \Gamma \vdash e\ [\vec{\rho}]\ e' : \tau', \{\rho, \epsilon\} \cup \varphi_0 \cup \varphi \cup \varphi'}\ [\text{T-FunApp}]$$

Fig. 2. Effect Typing Rules for Values and Expressions.

for values. In expressions $\mathtt{letregion}\ \{\vec{\rho}\vec{\epsilon}\}\ \mathtt{in}\ e$, the variables $\vec{\rho}$ and $\vec{\epsilon}$ are *bound* in e. We consider expressions identical up to renaming of bound names.

3.5 Typing Rules

Environments (Γ) map program variables to type schemes. When Γ and Γ' are environments, we write $\Gamma + \Gamma'$ to denote the environment with domain $\mathrm{Dom}\,\Gamma \cup \mathrm{Dom}\,\Gamma'$ and values $(\Gamma + \Gamma')(x) = \Gamma(x)$ if $x \in \mathrm{Dom}\,\Gamma$ and $(\Gamma + \Gamma')(x) = \Gamma'(x)$ if $x \notin \mathrm{Dom}\,\Gamma$.

The typing rules for values and expressions are mutually dependent and are given in Figure 2. The typing rules for values allow inference of sentences of the form $\vdash v : \sigma$, stating that "the value v has type scheme σ". The typing rules for expressions allow inference of sentences of the form $C, \Gamma \vdash e : \sigma, \varphi$, which states that "under the constraint set C and in the type environment Γ, the expression e has type scheme σ and effect φ.

There are several aspects to note about the typing rules. First, in the typing rules for functions, the latent effect of functions are annotated as arrow effects on the arrows. We see here the importance of arrow effects, which allows us to identify effects comprised of other effects.[3] Second, both the value and expression rules for recursive functions allow for polymorphic recursion in regions and effects, whereas polymorphic recursion in types is not supported. Third, in the typing rule for **letregion**, we assert that created local regions (and effects) are distinct from all other region and effect variables. Fourth, the typing rule for **assert** expressions requires that the asserted basic constraint is entailed by the constraint set provided by the context. Recall here that the constraint entailment also provides a guarantee that the constraint is valid. Also, constraint entailment ensures that the region and effect variables occurring in the involved constrained sets in **assert** expressions are free in the constraint set provided by the context. In particular, the rules [TV-Fun] and [T-Fun] introduce nil-constraints in the typing judgments for the body of the recursive function, which make the abstracted region and effect variables available for assertions (the **assert** construct requires involved variables to occur free in the contextual constraint set). Another observation is that free region- and effect-variables occurring in an asserted basic constraint in the **assert** construct are added to the effect set of the **assert** expression, which ensures that the involved regions and effects are not being discharged prematurely by invocation of the **letregion** evaluation rule. Finally, the typing rule for recursive function calls requires that such function calls are fully applied, which models that recursive function calls need not lead to intermediate closures; instead, both region arguments and the value argument can be provided in registers (or in a stack frame).

3.6 Typing Properties

The typing rules possess some important properties that we shall emphasise. First, the typing rules are closed under environment and constraint extensibility:

Proposition 3.3 (Environment and Constraint Extensibility). *If* $C, \Gamma \vdash e : \pi, \varphi$ *then* $C' \cup C, \Gamma' + \Gamma \vdash e : \sigma, \varphi$ *for any* C' *and* Γ'.

Proof. By straightforward induction on the derivation of $C, \Gamma \vdash e : \sigma, \varphi$ using the property of constraint entailment extensibility. □

The typing rules are also closed under value substitution:

Proposition 3.4 (Value Substitution). *If* $C, \Gamma + \{x : \sigma\} \vdash e : \sigma', \varphi$ *and* $C \vdash v : \sigma$ *then* $C, \Gamma \vdash e[v/x] : \sigma', \varphi$.

Proof. By induction on the derivation of $C, \Gamma + \{x : \sigma\} \vdash e : \sigma', \varphi$. □

Finally, the typing rules are also closed under well-constrained substitution, which is demonstrated by induction on the typing derivation:

Proposition 3.5 (Typing Closed Under Substitution). *If* $C \cup C', \Gamma \vdash e : \sigma, \varphi$ *and* $C \models S \bullet C'$ *then* $C \cup S(C'), S(\Gamma) \vdash S(e) : S(\sigma), S(\varphi)$.

Proof. By induction on the derivation of $C \cup C', \Gamma \vdash e : \sigma, \varphi$. The proof makes essential use of Proposition 3.2 for the case of recursive function application and essential use of Proposition 3.1 for the case of assertions. □

[3] If we had instead annotated arrows with effects of the form $\{\epsilon, \varphi\}$, given two such effects, $\{\epsilon, \varphi\}$ and $\{\epsilon', \varphi'\}$, we would not be able to come up with a unifier (i.e., a substitution) that would uniquely identify the two effects as other effect variables could appear in φ and φ'.

Allocation and Deallocation $\boxed{e \xhookrightarrow{\varphi} v}$

$$\begin{array}{rcl} \lambda x.e \ \texttt{at}\ \rho & \xhookrightarrow{\{\rho\}\cup\varphi} & \langle \lambda x.e \rangle^{\rho} \\ \texttt{fun}\ f : \sigma\ [\vec{\rho}]\ x = e \ \texttt{at}\ \rho & \xhookrightarrow{\{\rho\}\cup\varphi} & \langle \texttt{fun}\ f : \sigma\ [\vec{\rho}]\ x = e \rangle^{\rho} \\ \texttt{letregion}\ \varphi' \ \texttt{in}\ v & \xhookrightarrow{\varphi} & v \end{array}$$

Reduction and Context $\boxed{e \xhookrightarrow{\varphi} e'}$

$$\begin{array}{rcll} \langle \lambda x.e \rangle^{\rho}\ v & \xhookrightarrow{\{\rho\}\cup\varphi} & e[v/x] & \\ \langle \texttt{fun}\ f : \sigma\ [\vec{\rho}]\ x = e \rangle^{\rho}\ [\vec{\rho}']\ v & \xhookrightarrow{\{\rho\}\cup\varphi} & e[\vec{\rho}'/\vec{\rho}][v/x][\langle \texttt{fun}\ f : \sigma\ [\vec{\rho}]\ x = e \rangle^{\rho}/f] & \\ \texttt{let}\ x = v\ \texttt{in}\ e & \xhookrightarrow{\varphi} & e[v/x] & \\ \texttt{assert}\ \varphi' \#\ \varphi''\ \texttt{in}\ e & \xhookrightarrow{\varphi} & e & (\varphi' \cap \varphi'' = \emptyset) \end{array}$$

$$E_{\varphi}[e] \xhookrightarrow{\varphi'} E_{\varphi}[e'] \quad \text{if } e \xhookrightarrow{\varphi\cup\varphi'} e' \text{ and } E_{\varphi} \neq [.] \text{ and } \varphi \cap \varphi' = \emptyset \quad [\textsc{Ctx}]$$

Fig. 3. Dynamic semantics for region-annotated programs.

3.7 Dynamic Semantics

To give a dynamic semantics for the region-annotated language, we first define the grammar for *redexes* (r) and *evaluation contexts* (E_{φ}):

$$\begin{array}{rcll} r & ::= & \texttt{letregion}\ \varphi\ \texttt{in}\ v \mid \texttt{assert}\ c\ \texttt{in}\ e & \\ & \mid & \texttt{let}\ x = v\ \texttt{in}\ e \mid \langle \lambda x.e \rangle^{\rho}\ v \mid \langle \texttt{fun}\ f : \sigma\ [\vec{\rho}]\ x = e \rangle^{\rho}\ [\vec{\rho}']\ v & \\ & \mid & \lambda x.e\ \texttt{at}\ \rho \mid \texttt{fun}\ f : \sigma\ [\vec{\rho}]\ x = e\ \texttt{at}\ \rho & \\ E_{\varphi} & ::= & [.] & \\ & \mid & \texttt{letregion}\ \varphi''\ \texttt{in}\ E_{\varphi'} & (\varphi = \varphi'' \cup \varphi') \\ & \mid & \texttt{let}\ x = E_{\varphi}\ \texttt{in}\ e \mid E_{\varphi}\ e \mid v\ E_{\varphi} & \\ & \mid & E_{\varphi}\ [\vec{\rho}]\ e \mid v\ [\vec{\rho}]\ E_{\varphi} & \end{array}$$

Evaluation contexts E_{φ} make explicit, through φ, the region and effect variables bound to regions and effects in encapsulating `letregion` constructs. When E_{φ} is an evaluation context and e is an expression, we write $E_{\varphi}[e]$ to denote the expression formed by filling the hole $[.]$ in the context E_{φ} with the expression e.

The evaluation rules are given in Figure 3 and consist of *allocation and deallocation rules*, *reduction rules*, and a *context rule*. The rules are of the form $e \xhookrightarrow{\varphi} e'$, which says that, given a set of allocated regions φ, the expression e reduces to the expression e' in one step. Notice the reduction rule for `letregion` constructs, which puts no constraints on which regions are deallocated. In contrast, the reduction rule for `assert` is instrumented with the requirements that the two involved constrained effect sets are disjoint.

We further define the *evaluation* relation $\xhookrightarrow{\varphi}{}^{*}$ as the least relation formed by the reflexive transitive closure of the relation $\xhookrightarrow{\varphi}$. We further define $e \Downarrow_{\varphi} v$ to mean $e \xhookrightarrow{\varphi}{}^{*} v$, and $e \Uparrow_{\varphi}$ to mean that there exists an infinite sequence, $e \xhookrightarrow{\varphi} e_1 \xhookrightarrow{\varphi} e_2 \xhookrightarrow{\varphi} \cdots$.

3.8 Type Safety

The proof of type safety is based on well-known techniques for proving type safety for statically typed languages [Morrisett 1995; Wright and Felleisen 1994].

A well-typed expression is either a value or it can be expressed as the composition of an evaluation context and a redex.

Proposition 3.6 (Unique Decomposition). *If $C \vdash e : \sigma, \varphi$, then either (1) e is a value, or (2) there exist a unique $E_{\varphi'}$, e', and σ' such that $e = E_{\varphi'}[e']$ and $C \vdash e' : \sigma', \varphi \cup \varphi'$ and e' is a redex.*

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 3.7 (Type Preservation). *If $C \vdash e : \sigma, \varphi$ and $e \xhookrightarrow{\varphi} e'$ then $C \vdash e' : \sigma, \varphi$.*

Proof. By induction on the derivation $e \xhookrightarrow{\varphi} e'$. The most involved cases include the case for contextual evaluation (i.e., [Ctx]), which proceeds by case analysis on the structure of contexts, and the case for recursive function application.

Case $e = \langle \mathbf{fun}\ f : \sigma'\ [\vec{\rho}]\ x = e \rangle^{\rho}\ [\vec{\rho}']\ v$. We have $e' = e[\vec{\rho}'/\vec{\rho}][v/x][\langle \mathbf{fun}\ f : \sigma'\ [\vec{\rho}]\ x = e\rangle^{\rho}/f]$ and $\rho \in \varphi$. The desired result is established using Proposition 3.4 and Proposition 3.5.

The remaining cases are straightforward. □

Proposition 3.8 (Progress). *If $C \vdash e : \sigma, \varphi$ then either e is a value or $e \xhookrightarrow{\varphi} e'$, for some e'.*

Proof. If e is not a value, then by Proposition 3.6 there exist a unique $E_{\varphi'}$, r, and σ' such that $e = E_{\varphi'}[r]$ and $C \vdash r : \sigma', \varphi \cup \varphi'$. The remainder of the proof argues that $r \xhookrightarrow{\varphi \cup \varphi'} e_2$, for some e_2, so that $E_{\varphi'}[r] \xhookrightarrow{\varphi} E_{\varphi'}[e_2]$ follows from [Ctx] in Figure 3. We proceed by case analysis.

Case $r = \mathbf{letregion}\ \varphi'\ \mathbf{in}\ v$, for some φ' and some v. We have immediately, from the region reduction rule, that $e_2 = v$.

Case $r = \mathbf{assert}\ c\ \mathbf{in}\ e''$, for some c and e''. From the typing rule for assertions, we have $C \models \{c\}$ and thus $\vdash c$ holds. We know $c = \varphi' \# \varphi''$, for some φ' and φ'', and because $\vdash c$ holds, it follows directly that $\varphi' \cap \varphi'' = \emptyset$. We can now apply the reduction rule for assertions to get $e_2 = e''$.

The remaining rules follow trivially. □

Theorem 3.9 (Type Soundness). *If $\vdash e : \sigma, \varphi$, then either $e \Uparrow_{\varphi}$ or $e \Downarrow_{\varphi} v$ and $\vdash v : \sigma, \varphi$, for some v.*

Proof. By induction on the length of the evaluation sequence, applying Proposition 3.7 and Proposition 3.8. □

4 SYNTACTIC CONSTRUCTS

As mentioned earlier, a Standard ML program is also a ReML program. ReML introduces new name spaces for explicit region variables and explicit effect variables. Besides the top-level region variables `r0top`, `r0pair`, `r0triple`, `r0ref`, `r0array`, and `r0string` and the top-level effect variable `e0`, new explicit region and effect variables may be introduced using **`with`** declarations (in which variables prefixed with an `e` are deemed to be effect variables). Here is an example function that declares a local region `r` for storing a temporary pair, followed by a projection of the first element:

```
fun f () : int = let with r
                     val x = (3,5)`r
                 in #1 x
                 end
```

Notice how the pair is annotated with an explicit region-annotation, specifying that the pair should be stored in the local region `r`. As an alternative, we may choose that the pair should be stored in the global region `r0pair`:

```
fun g () : int = let val x = (3,5)`r0pair
                 in #1 x
                 end
```

Either way, ReML decides that the programs are well-typed according to the region-inference typing rules, albeit the latter function `g` will leak a pair whenever the function is called, a property, which (we shall see) is captured in the effect of the function.

An alternative to annotating allocating expressions with region information, a ReML programmer may use region-annotated type ascriptions to enforce values to be stored in particular regions:

```
fun h () : int = let val x : (int * int)`r0pair = (3,5)
                 in #1 x
                 end
```

In general, ReML will happily optimise (i.e., simplify) the program [Elsman and Hallenberg 1995], but if optimisations are disabled, we can get ReML to print the internal representation of `h`:

```
fun h at r0top [] (v86) =
   let val x = (3, 5)at r0pair
       with r15:1
   in  (fn at r15 v90 ⇒ let val v91 = #1 v90
                        in v91
                        end) x
   end
```

One aspect to notice is that an internal region analysis [Birkedal et al. 1996] has determined that the region `r15` takes up only one word of memory (for storing the code pointer of the closure), which means that `r15` will be allocated in the function frame of `h`. We shall later in Section 6 discuss extensions that make it possible to specify such properties directly as programmer annotations.

Instead of storing values in local or global regions, a function may store values in regions that are passed as parameters to the function. As a simple example, here is a function that takes an integer as argument and creates a list of integers:

```
fun down `r (n:int) : int list`r =     (* down n = [n,...,2,1] *)
   case n of
      0 ⇒ nil
    | _ ⇒ n :: down (n-1)
```

When calling the function, the programmer may choose to be explicit about region parameters or rely on region inference to do its best (as in the recursive call to `down` above). Assuming a function `first:int list→int`, returning `~1` (negative one) if the argument list is empty, here is an example call to `down`, followed by a call to `first`:

```
val x = let with r
        in first (down `r 5)
        end
```

Here, ReML has inferred that it is safe to deallocate the local region `r` after extracting the first element of the list.

4.1 Exomorphisms and Endomorphisms

An *exomorphic* function is a function that stores its result in regions different from those containing the arguments. Here is an example exomorphic function:

```
fun copy `[r1 r2] (xs : α list`r1) : α list`r2 =
   case xs of
      nil ⇒ nil
    | x :: xs ⇒ x :: copy xs  (* If we forget copy, ReML complains! *)
```

Here we have specified that `copy` receives its argument in `r1` and returns a list in `r2`. If we had forgotten to copy `xs` in the last line, ReML complains with an error message:

```
copylist.sml, line 4, column 9:
  fun copy `[r1 r2] (xs : α list`r1) : α list`r2 =
           ^^^^^^^^
Cannot unify the explicit region variables `r1 and `r2
```

An *endomorphic* function, on the other hand, is a function that stores its result in the same regions as its arguments. A good example is the infix append-function `@`, which is defined as follows:

```
fun op @ `[r1 r2] (xs : α list`r1, ys: α list`r2) : α list`r2 =
   case xs of
      nil ⇒ ys
    | x :: xs ⇒ x :: (xs @ ys)
```

Notice that the first argument list is allowed to reside in a different region than the second argument and that the result is stored in the same region as the second argument. If we want to make sure that the result is stored into a region different from the second argument, the programmer may first copy the second argument:

```
fun copyappend `[r1 r2 r3] (xs:α list`r1,ys:α list`r2) : α list`r3 =
  xs @ (copy ys)
```

It is an important design choice that explicit region variables are never unified, which gives us modular properties about functions that are explicitly annotated. Notice, however, that a calling context may pass the same region for different formal explicit region parameters. We shall see later, in Section 5, that ReML allows for expressing that two arguments reside in different regions.

4.2 Specifying Effects

ReML distinguishes between explicit region variables and explicit effect variables, both of which may be declared using **`with`** declarations and as parameters to functions (using the `` ` `` notation).

Here is an example of a type ascription that refers to an explicit effect variable:

```
val x = let with e
            val f : int #e→int = fn x ⇒ x+8
        in f 3
        end
```

Whereas explicit effect variables are often not useful in their own right, they are important for expressing constraints through the use of **while**-types, as we shall see in Section 5.

5 EFFECT CONSTRAINTS

Effect constraints, which is a novel concept in ReML, are expressed using so-called **while**-types, which are types annotated with the **while** keyword, followed by an effect constraint. ReML supports effect constraints on the forms e_1 `##` e_2 and e_1 `#` e_2, where e_1 and e_2 are effects. An effect in ReML is either an effect variable or a set of atomic effects. An atomic effect takes one of the forms `get r`, `put r`, or `e`, where `r` is an explicit region variable and `e` is an explicit effect variable (in the formalisation in Section 3, atomic effects of the forms `put r` and `get r` are conflated as the atomic effect ρ). Multiple constraints may be specified using nested **while**-types.

Constraints of the form e_1 `##` e_2 express that e_1 and e_2 contain no intersecting put-effects. Constraints of the form e_1 `#` e_2 express that the set of explicit region and effect variables of e_1 and e_2 are disjoint.

5.1 Unlocking Parallelism

In recent work, a fork-join parallel construct has been added to the MLKit [Elsman and Henriksen 2023]. In essence, the fork-join functionality is expressed through the following interface:

```
structure Thread : sig
  type α t
  val spawn : (unit→α) → (α t→β) → β
  val get : α t → α
end
```

Here, the `spawn` function takes two functions as arguments. In a call `spawn f g`, the function `f` is evaluated in parallel with `g`, which may use the `get` functionality to join the thread executing `f`.

A particular problem that the `spawn` interface exposes is that there are no guarantees that the allocation effect of evaluating `f` does not intersect with the allocation effect of evaluating `g`. In particular, the two threads may each attempt to modify a shared allocation pointer, causing a race-condition, unless a mutual exclusion lock (i.e., a mutex) is used for controlling access to the shared resource.

Whereas the overhead of potential race-conditions may be mitigated with a so-called protection inference [Elsman and Henriksen 2023], in ReML, we can express that the allocation effects of the two threads are disjoint, using **while**-types, as is also demonstrated in Section 2:

```
val spawn : (unit #e1→ α) → (α t #e2→ β) → β
            while e1 ## e2
```

Here the effect variables `e1` and `e2` are implicitly quantified in function specifications. While **while**-types allow for expressing effect constraints of the kind shown for `spawn`, it now becomes the obligation of the caller to establish that the effects of the two provided functions do not have intersecting allocation effects. In particular, instantiating `spawn` in some context must obey the **while**-type constraint, which may require evidence obtained from other **while**-type constraints.

Here is how to implement MPL's `par` function:

```
fun par (f: unit #e1→ α) (g: unit #e2→ β) : α * β while e1 ## e2 =
  spawn g (fn t ⇒ (f(),get t))
```

6 EXTENDING THE NOTIONS OF EFFECTS

Although ReML provides general support for managing effects through its type- and effect-system, each type of effect must be added to ReML case-by-case. ReML currently supports effects that are related to memory allocation and memory use through the notion of `put` and `get` effects, but the foundation supports other kinds of effects and refinements of memory effects, as we shall discuss in detail below.

6.1 Refinement of Memory Effects

Allowing programmers to reason about mutable updates at the type level provides a good foundation for reasoning about local state. We refine atomic effects and effect constraints as follows:

$$\begin{array}{rcl} \eta & ::= & \epsilon \mid \mathtt{put}\ \rho \mid \mathtt{get}\ \rho \mid \mathtt{mut}\ \rho \\ c & ::= & \varphi \,\#\, \varphi' \mid \varphi \,\#\#\, \varphi' \mid \mathtt{noput}\ \varphi \mid \mathtt{nomut}\ \varphi \end{array}$$

Notice that the **`letregion`** construct (`with` declaration in ReML) acts as a handler for mutation effects in a local region. Thus, local mutable state is supported and mutation cannot be observed if the region holding the mutable data is local to the function.

In practice, here are the types for reference construction, dereference, and assignment, without showing auxiliary region and effect variables:

$$\begin{array}{l} \mathtt{val\ ref} \quad : \alpha \xrightarrow{\{\mathtt{put}(\rho)\}} (\alpha\ \mathtt{ref},\ \rho) \\ \mathtt{val\ op\ :=} : (\alpha\ \mathtt{ref}, \rho) * \alpha \xrightarrow{\{\mathtt{get}(\rho),\mathtt{mut}(\rho)\}} \mathtt{unit} \\ \mathtt{val\ !} \quad : (\alpha\ \mathtt{ref},\ \rho) \xrightarrow{\{\mathtt{get}(\rho)\}} \alpha \end{array}$$

We see that the latent effect specified for mutable update (`:=`) includes a $\mathtt{mut}(\rho)$ effect. As for reference updates, the type for array update (`Array.update`) contains a `mut` effect on the region containing the array.

Notice also the support for effect constraints on the form $\mathtt{noput}(\epsilon)$ and $\mathtt{nomut}(\epsilon)$, which allow for specifying that a function does not allocate in non-local regions and that a function makes no mutations in non-local data structures. Moreover, with the refinement of atomic effects, it is natural to include a basic constraint $\varphi \,\#\#\, \varphi'$, which specifies that `put` effects in φ and φ' do not intersect. For the formalisation, we extend the notion of what constitutes a valid basic constraint (i.e., the relation $\vdash c$):

Basic Constraint Validity for Refined Memory Effects $\boxed{\vdash c}$

$$\frac{\nexists \rho.\mathtt{mut}\ \rho \in \varphi}{\vdash \mathtt{nomut}\ \varphi} \qquad \frac{\nexists \rho.\mathtt{put}\ \rho \in \varphi}{\vdash \mathtt{noput}\ \varphi} \qquad \frac{\nexists \rho.\mathtt{put}\ \rho \in (\varphi \cap \varphi')}{\vdash \varphi \,\#\#\, \varphi'} \qquad \frac{\mathrm{frev}\ \varphi \cap \mathrm{frev}\ \varphi' = \emptyset}{\vdash \varphi \,\#\, \varphi'}$$

We further refine the notion of normalisation to contain also definitions for the new cases. Thus, we have, for instance, $||\mathtt{nomut}\ \varphi|| = \{\mathtt{nomut}\ \{\eta\} \mid \eta \in \mathrm{fev}\ \varphi \vee \exists \rho.\eta = \mathtt{mut}\ \rho \wedge \eta \in \varphi\}$.

It is important to notice here that validity of a basic constraint is not a sufficient property for the basic constraint to be satisfied. In order for a basic constraint to be satisfied, it must be entailed by the constraint set specified in the context of a particular call site, which may include precondition constraints for effect variables, for instance, that have not yet been instantiated.

A further possibility would be to allow the programmer to be explicit about whether a region should be allocated directly on the stack, which it can be if it can be determined statically that only one value will ever reside in the region and that the maximum size of that value can be determined statically [Birkedal et al. 1996]. Constraining function to allocate only on the stack

may be a desirable property of a function and it may be desirable for a programmer to specify and reason about such a property.

6.2 Exceptions

Extending the effect language to allow also for specifying possibly uncaught exceptions [Fahndrich et al. 1998; Pessaux and Leroy 1999] fit naturally in ReML. Special care must be taken to deal properly with the generative nature of exceptions in Standard ML (and ReML), however.[4] A key benefit in ReML from integrating region inference and exception inference is that exceptions that can be inferred to live locally may be allocated in local regions. Currently, exceptions that carry payloads are allocated in global regions to ensure that exception-carried values are still allocated at potential handler-sites.

In the following, we use n to range over exception names. For extending the ReML formalism with support for exceptions (that do not take payloads), we first extend the language with support for raising exceptions and for handling exceptions, as well as typing rules for specifying the typing and effect properties for the new constructs. For simplicity, we assume that the construct for raising an exception takes an exception name as argument and that the handling construct has only one branch for handling a particular exception:

$$\frac{}{C, \Gamma \vdash \texttt{raise}\ n : \tau, \{\mathsf{exn}(n)\}} \qquad \frac{C, \Gamma \vdash e : \tau, \varphi \quad C, \Gamma \vdash e' : \tau, \varphi'}{C, \Gamma \vdash e\ \texttt{handle}\ n \Rightarrow e' : \tau, (\varphi \setminus \{\mathsf{exn}(n)\}) \cup \varphi'}$$

The dynamic semantics is extended to allow for an expression to evaluate to an exception and particular context rules are added for formally defining the propagation of raised exceptions (we shall not give the rules here).

We then refine the notion of atomic effects and basic constraints as follows:

$$\begin{array}{rcll} \eta & ::= & \ldots & \mid\ \mathsf{exn}\ n \\ c & ::= & \ldots & \mid\ \mathsf{noexn}\ \varphi \end{array}$$

We further extend the notion of what constitutes a valid basic constraint (i.e., the relation $\vdash c$), as we did for the refined memory effects:

Basic Constraint Validity for Exceptions $\boxed{\vdash c}$

$$\frac{\nexists n.\mathsf{exn}\ n \in \varphi}{\vdash \mathsf{noexn}\ \varphi}$$

Finally, we further refine the notion of normalisation to contain also a definition for the new case. Thus, we have $||\mathsf{noexn}\ \varphi|| = \{\mathsf{noexn}\ \{\eta\} \mid \eta \in \mathsf{fev}\ \varphi \vee \exists n.\eta = \mathsf{exn}\ n \wedge \eta \in \varphi\}$.

Notice that for normalisation, we eliminate only those atomic effects that may not result in invalid constraints.

6.3 Exotic Effects

Other relevant effects that may be modeled in ReML include non-termination, non-determinism, and IO effects.

For IO effects, we can simply add an atomic effect `io` and enrich each library function that perform IO by adding the `io` atomic effect to the latent effect of the function type. Such effects will have no natural handler but can be used to reason about whether a function is pure, for instance, in conjunction with a basic effect constraint on the form `noio` φ.

[4]The notion of generative exceptions allows exceptions to be declared with type variables that are bound at higher levels. For soundness, a new "dynamic name" is generated whenever an exception is declared and it is thus not always known statically at exception-handler sites which exception is handled.

For reasoning about non-termination, we may add an atomic effect `rec` to the language along with a basic effect constraint on the form `norec` φ. In this way, we can reason about termination in a structural way, which may also be used to determine that a function as pure, which, again, can be used by a compiler optimiser pass. By further distinguishing between recursive and tail-recursive function calls (refine atomic effects of the form `rec` into the atomic effects `trec` and `rec`), we may add support in ReML for giving guarantees that a function runs in bounded space (no heap allocation and no stack allocation). Again, as for `io` effects, termination and boundedness effects will have no natural handler.

There are quite a few other possibilities. For instance, ATS [Chen and Xi 2005] makes it possible to distinguish between functions that are represented as closures (which have a non-empty environment) and functions that need no environment. We can model such properties with atomic effects specifying whether the function will read from a closure or not when evaluated.

It would be interesting to explore useful predicates over effects, including a purity predicate, a non-allocation predicate, and an idempotence predicate. Such predicates could potentially be used for implementing optimisations in the backend part of ReML.

One limitation of the ReML effect system is that it does not support reasoning about linear or affine use of resources, as effects are really modeled as sets of atomic effects and because a function can be annotated with any arbitrary effect (meaning that the function could potentially have this effect). While this restriction is well suited for inference and typing, a possibility for future work would be to support effects that have more substructural properties.

7 INTEGRATION OF REGION INFERENCE AND CONSTRAINT RESOLUTION

Formally, region inference can be presented as a substitution-based type-inference algorithm, based on Hindley-Milner's algorithm W and augmented with a mechanism for discharging effects and regions that are local to an expression (for insertion of **`letregion`** constructs). Due to the support for polymorphic recursion in regions and effects, the algorithm is split into a so-called *spreading phase*, which annotates allocation sites with fresh regions and function types with effects identified by fresh effect variables. A separate co-called *contraction phase* applies *contracting substitutions* (i.e., substitutions that unify effects and regions) repeatedly in order for a program to adhere to the region typing-rules. In the contraction phase, no new effect and region variables are created, thus, the algorithm is guaranteed to terminate with a region-annotated program that satisfies the region typing-rules.

ReML augments region-inference with syntax (explicit region- and effect-variables) for referring to the underlying region- and effect-variables that region inference works with. Substitution is implemented as graph-unification, with unifiable nodes being region- and effect-variables and edges representing effect membership (the implementation also represents other atomic effects, such as `mut`(ρ), `put`(ρ), and `get`(ρ) as nodes). Here is an overview of how the different ReML annotation mechanisms influence region inference, which, without annotations, is guaranteed to result in a well-typed program, provided the underlying program is a well-typed Standard ML program:

(1) During the spreading phase, explicitly annotated regions (annotated using the back-tick syntax) are looked up in the environment when spreading expressions and types. The implementation complaints with a type error if unification attempts to unify two different explicit effect variables or two different explicit region variables.
(2) Explicit **`with`** declarations and support for explicit region- and effect-parameters may be used to pin region- and effect-variables to a particular scope. This pinning is implemented by introducing fresh region- and effect-variables and binding them to the explicit counterparts in the environment used for spreading subexpressions. If unification results in a violation

of the pinning (that is, if region inference is forced to push the binding outwards), a type error is reported. For details about how effects are unified, we refer the reader to [Tofte and Birkedal 1998] and [Tofte and Birkedal 2000], in particular with respect to information about how to deal with so-called *secondary* region- and effect-variables, variables that occur in arrow effects but are not paired directly with a type constructor.

(3) Effect constraints, expressed using the **`while`**-type mechanism, have no influence on region inference, which, as we have seen, is in contrast to explicit **`with`** annotations, explicit parameter annotations, and explicit `at`-annotations. Instead, all effect constraints are checked after region-inference, where we can assume that all graph cycles have been collapsed. The implementation decomposes effect constraints into basic constraints that are added to the nodes of the effect-graph. When parts of the graph are copied, which happens when a type scheme is instantiated, also the constraints are copied (and instantiated). For discharging an effect, it is sufficient to check each basic constraint, under the assumption that other constraints are satisfied (cycles in the graph have been collapsed by region inference).

From a formal perspective, region inference makes the assumption that the set of arrow effects occurring in a region- and effect-annotated program is *consistent*, meaning that the set is *functional* (i.e., each effect variable is associated with a unique effect), *transitive* (i.e., if $\epsilon.\varphi$ and $\epsilon'.\varphi'$ are both in the set then $\epsilon' \in \varphi$ implies $\varphi' \subseteq \varphi$), and *closed* (i.e., each effect variable occurring in the set is defined by the set) [Tofte and Birkedal 1998]. Whereas this consistency property is not used for establishing soundness, it is essential for establishing termination and correctness of region inference. A central property of a contracting substitution is that when it is applied to a consistent set of arrow effects, the result is itself a consistent set of arrow effects. We see here the link to the implementation that uses unifiable graphs for representing arrow effects. Much like Hindley-Milner's algorithm W, where type substitution may be implemented by unification, region inference may be implemented using region- and effect-unification to model region- and effect-substitutions.

8 REML STATUS AND LARGER EXAMPLES

Whereas ReML is a fully working system featuring explicit region and effect annotations, `mut`, `put`, and `get` effects, and constraints such as `noput`, `nomut`, and `##` constraints, current work aims at improving type error reporting with proper indication of the source of a constraint violation. The changes to the implementation have little influence on overall compilation times (if any) as overhead is introduced only in relation to managing annotations, effect constraint propagation, and constraint checking.

We have carried out experiments on larger benchmarks from [Elsman and Henriksen 2023], including the benchmarks `mandelbrot`, `vpmsort`, `pmsort`, and `ray`. Whereas the benchmark programs `vpmsort` and `pmsort` uses the `par` function shown in the paper, the benchmarks `mandelbrot` and `ray` uses a `parfor` function, which, as `par`, is based on the underlying `spawn` functionality, and which applies a function of type `int → unit` in parallel on an interval of integers. Using a `noput` constraint, the `##` constraint on the used `spawn` function can be discharged and we get the property that the ray-tracer and Mandelbrot image creator can execute without allocation races. In the process of porting the `ray` benchmark, we were directed by the type system to modify the ray-tracer implementation (using an array-of-structs to struct-of-arrays transformation) to ensure that pixel values (triples of integers) are not allocated by the individual threads but rather saved in individual channel arrays.

ReML is open source and is available from the MLKit source code repository.[5] Moreover, this paper comes with an artifact, which, in addition to a snapshot of the ReML source code, includes a

[5] See http://github.com/melsman/mlkit.

tutorial demonstrating the features of ReML presented in this paper, and an in-depth description of the implementation aspects of ReML [Elsman 2024]. ReML builds on the MLKit infrastructure and shares source code with the native MLKit backend, targeting x68-64 machine code, and SMLtoJs [Elsman 2011], which targets JavaScript engines.

The artifact is comprised by a docker image, which readily makes available the ReML compiler in terms of a `reml` executable. The `reml` executable accepts ReML programs as input and generates x86-64 machine code as a result. The artifact tutorial demonstrates how to apply `reml` to the examples presented in this paper and how to apply ReML to new examples. Concretely, the artifact describes the ReML parallel library and demonstrates how to compile and run the parallel Mergesort example, the parallel Mandelbrot example, and the parallel ray-tracer.

9 RELATED WORK

Much related to this work is the work on Cyclone [Gerakios et al. 2010; Grossman et al. 2002; Hicks et al. 2004; Swamy et al. 2006], a C-like programming language aimed at safe system-level programming (and even thread-based programming) based on regions, but with limited support for region inference and higher-order programming. Another region-based language is RC [Gay and Aiken 1998], which features support for explicit regions in C, combined with reference counting of regions (instead of relying on the stack discipline). Another related language for system-level programming is ATS [Chen and Xi 2005], which, in addition to supporting refinement types and dependent types, also allows a programmer to specify specific properties about a function, such as the property that a function has no free variables and therefore can be represented without a closure and only by its code pointer.

The region- and effect calculus by Tofte and Talpin [Tofte and Talpin 1997] can also be modeled using polymorphism and monads [Fluet and Morrisett 2004], which has inspired extensions to region-based memory management leading, for instance, to a discipline that does not follow a LIFO-lifetime of regions, and uses of region-based memory management for managing other types of resources, such as file descriptors [Kiselyov and Shan 2008]. The notion of monadic regions has also contributed with techniques for reasoning about code generation in multi-stage languages [Kiselyov et al. 2016]. In comparison to the techniques we present here, monadic regions are centralised around an "outlives" relationship between regions, suggesting that one region lives longer than another, and a subset relation between effects. Whereas these relations induce a notion of subtyping constraints, region inference in ReML is based on substitutions with arrow effects providing the mechanism that allows for finding region- and effect-unifiers (i.e., substitutions) that will unify two arbitrary region- and effect-annotated types, as long as their underlying ML types (i.e., after region- and effect-erasure) are identical. We shall not here argue that the one approach is better than the other, but rather emphasise that the design span here is large, which is also exemplified by the existence of two different inference algorithms for region-inference [Birkedal and Tofte 2001; Tofte and Birkedal 1998].

Besides the work on monadic regions, there has been a large body of work on providing simpler proofs of soundness for region-based systems, including [Calcagno 2001; Calcagno et al. 2002; Helsen and Thiemann 2001], and [Elsman 2023], which consider soundness in the context of augmenting region-based memory management with reference-tracing garbage collection. The present work builds on these techniques, which are all based on the strategy of syntactic soundness proofs [Morrisett 1995; Wright and Felleisen 1994].

Also related to the present work is the work on Embedded ML [Pareto 2000], which is inspired by the intermediate region-based program representation in the MLKit. Contrary to ReML, Embedded ML aims at more precise reasoning about the sizes of allocated regions, while requiring the programmer to be explicit about regions (no region inference fallback). Related to this work involves

work on optimising the representation of regions, including distinguishing between regions that may be allocated directly on the stack and regions that are heap-allocated as a linked list of pages [Birkedal et al. 1996]. Region inference, as it is implemented in the MLKit (and ReML), may be combined with garbage collection [Elsman 2023; Elsman and Hallenberg 2021; Hallenberg et al. 2002]. This area of work is complementary to the present work, which may be used with and without reference-tracing garbage collection.

A dominant language that supports explicit programming with region-based memory is Rust [Jung et al. 2017; Klabnik and Nichols 2018], which owns much of its design to the earlier work on Cyclone, RC, previous work on region-based memory management, and capability-based memory management [Walker et al. 2000] and ownership types [Jung et al. 2017].

Another area of related work is the bulk of recent work on programming languages based on effect handlers, including Eff [Bauer and Pretnar 2015], Koka [Leijen 2014], Effekt [Brachthäuser et al. 2020b], and System C [Brachthäuser et al. 2022, 2020a]. Here both Effect, which also features an embedding in Scala, and System C are based on the notion of capabilities.

Yet another body of related work is the work on refinement types for refining the set of values being specified by a type. Such work includes the seminal work on refinement types for ML [Freeman and Pfenning 1991] and newer work on refinement types in Liquid Haskell [Vazou et al. 2014]. We consider combining the type- and effect-system of ReML with a more traditional refinement type system a good candidate for future work.

One of the motivating examples of the present work is efficient support for parallelism. The present work uses much of the infrastructure presented in [Elsman and Henriksen 2023] for parallelism support in ReML, with the difference that in ReML, protection against allocation races can be ensured statically with the use of **`while`**-types. In contrasts, the constraint-based protection-inference algorithm, presented in [Elsman and Henriksen 2023], aims at distinguishing regions that require protection (e.g., using mutexes) from those that do not. With the lack of explicit constraints provided by the programmer, protection inference is fragile to small program changes and gives the programmer only limited control of the desired runtime behavior. Technically, protection inference infers, for each allocated region, whether a mutex should be associated with the region at runtime. Effect constraints in ReML and protection inference are complementary concepts in the sense that one mechanism does not necessarily preclude the other. It is perfectly fine to have two different versions of the `par` function around, one that uses protection inference and dynamic features, such as mutexes, to preclude allocation races, and another, that uses ReML effect constraints to give static guarantees about the avoidance of allocation races.

Much related work has investigated the possibilities for adding support for shared-memory parallel OS threads (and light-weight threads) in ML-like languages, such as OCaml [Sivaramakrishnan et al. 2020], Standard ML [Cooper and Morrisett 1990; Westrick et al. 2019], and Manticore [Farvardin and Reppy 2020; Fluet et al. 2008]. In all cases, the implementations require special attention to the garbage collection techniques used, in particular with respect to mutable effects. In the case of MPL [Westrick et al. 2019], which adds shared-memory fork-join parallelism to the MLton Standard ML compiler through a simple `par`-function, the memory discipline is centered around so-called *disentangled heaps*, for which each thread is associated with an individual heap and where pointers between heaps can only point upwards towards the root [Raghunathan et al. 2016]. The disentangled-heap property, which is required by MPL and its parallel garbage collection of leaf heaps, can be enforced automatically by a combination of static and dynamic techniques [Westrick et al. 2022]. If a thread assigns to an object allocated by a sibling, the object will be allocated in a memory region allocated by a common ancestor.

Finally, a large area of related work includes work on qualified types [Jones 1994], including qualifier inference for C [Foster et al. 2002], implementation of Haskell type classes [Hall et al. 1996;

Peterson and Jones 1993], elimination of polymorphic equality [Elsman 1998], and type inference with constraints, in general [Odersky et al. 1999; Sulzmann and Hudak 2000]. Whereas previous work shed important light on particular aspects of constraint inference and constraint abstraction, the present work differs from previous work by involving constraints on effects that are treated as separate objects in judgments. Much of the previous work on constraint solving, however, may show to carry over to the setting of effect constraints.

Another effect system based on boolean unification and constraints, is implemented in Flix. This system allows for tracking complementary effects, expressing, for instance, that an expression does not raise a particular exception [Lutze et al. 2023]. In Flix, however, it is not possible to express that two effects are disjoint.

10 CONCLUSION AND FUTURE WORK

We have presented ReML, a higher-order statically-typed functional language that allows programmers to be explicit about the effects performed by program code, including effects related to memory management. While work on ReML is ongoing, the current implementation is freely available, as described in Section 8.

There are a number of directions for future work. First, by integrating region inference and exception inference, exceptions that can be inferred to live locally may be allocated in local regions. Currently, exceptions that carry payloads are allocated in global regions to ensure that carried values are still allocated at potential handler-sites. Second, enriched effect information may open up for compiler optimisations that are otherwise difficult to implement. Such optimisations may involve reasoning about whether a function is pure (code elimination) or whether an effect is idempotent (code duplication).

With explicit annotations in ReML, it may be possible to augment region- and effect-inference with the possibility of higher-order region- and effect-polymorphism, existential regions [Henglein et al. 2001], and even first-class regions. We consider such developments good possibilities for future work.

We emphasise here that ReML is work-in-progress and that the integration of exception effects in the ReML effect system is under active development. That said, even in its current state, ReML can be used to reason locally about memory aspects of library functionality and for establishing guarantees about the lack of allocation races in parallel programs. We also consider it future work to investigate generalisations in terms of more general effect constraints (e.g., supporting set operations) and user-defined effects.

DATA AVAILABILITY STATEMENT

This paper is accompanied by a software artifact [Elsman 2024] that demonstrates the main contributions of the paper. A more detailed description of the content of the software artifact is available in Section 8.

ACKNOWLEDGMENTS

I would like to thank the anonymous reviewers for careful and constructive feedback on this work. I would also like to thank Troels Henriksen, Andrzej Filinski, and Ken Friis Larsen for many fruitful discussions leading to this paper. The work builds in essential ways upon the MLKit with Regions compiler infrastructure to which many people have contributed, including, in particular, Mads Tofte, Lars Birkedal, and Niels Hallenberg.

REFERENCES

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (jan 2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

Lars Birkedal and Mads Tofte. 2001. A constraint-based region inference algorithm. *Theoretical Computer Science* 258, 1 (2001), 299–392. https://doi.org/10.1016/S0304-3975(00)00025-6

Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. 1996. From Region Inference to von Neumann Machines via Region Representation Inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 171–183. https://doi.org/10.1145/237721.237771

Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, Capabilities, and Boxes: From Scope-based Reasoning to Type-based Reasoning and Back. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/3527320

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020a. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press. https://doi.org/10.1145/3428194

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020b. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *Journal of Functional Programming* 30 (2020), e8. https://doi.org/10.1017/S0956796820000027

Cristiano Calcagno. 2001. Stratified Operational Semantics for Safety and Correctness of the Region Calculus. In *ACM Symposium on Principles of Programming Languages (POPL'01)*. ACM Press.

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

Chiyan Chen and Hongwei Xi. 2005. Combining Programming with Theorem Proving. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia) *(ICFP '05)*. Association for Computing Machinery, New York, NY, USA, 66–77. https://doi.org/10.1145/1086365.1086375

E. Cooper and G. Morrisett. 1990. *Adding Threads to Standard ML.* Technical Report CMU-CS-90-186. Carnegie Mellon University, Department of Computer Science. Technical report.

Martin Elsman. 1998. Polymorphic Equality - No Tags Required. In *Proceedings of the Second International Workshop on Types in Compilation (TIC '98)*. Springer-Verlag, Berlin, Heidelberg, 136–155.

Martin Elsman. 2011. SMLtoJs: Hosting a Standard ML Compiler in a Web Browser. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients* (Portland, Oregon, USA) *(PLASTIC '11)*. Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/2093328.2093336

Martin Elsman. 2023. Garbage-Collection Safety for Region-Based Type-Polymorphic Programs. In *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI 2023)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3591229

Martin Elsman. 2024. Artifact for the POPL 2024 paper: Explicit Effects and Effect Constraints in ReML. Zenodo. https://doi.org/10.5281/zenodo.8425443

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).

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

Martin Elsman and Troels Henriksen. 2023. Parallelism in a Region-Inference Context. In *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI 2023)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3591256

Manuel Fahndrich, Jeffrey S. Foster, Jason Cu, and Alexander Aiken. 1998. *Tracking down Exceptions in Standard ML Programs.* Technical Report UCB/CSD-98-996. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/5561.html

Kavon Farvardin and John Reppy. 2020. From Folklore to Fact: Comparing Implementations of Stacks and Continuations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 75–90. https://doi.org/10.1145/3385412.3385994

Matthew Fluet and Greg Morrisett. 2004. Monadic Regions. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming* (Snow Bird, UT, USA) *(ICFP '04)*. Association for Computing Machinery, New York, NY, USA, 103–114. https://doi.org/10.1145/1016850.1016867

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2008. Implicitly-Threaded Parallelism in Manticore. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/1411204.1411224

Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-Sensitive Type Qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/512529.512531

Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 268–277. https://doi.org/10.1145/113445.113468

David Gay and Alex Aiken. 1998. Memory Management with Explicit Regions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 313–323. https://doi.org/10.1145/277650.277748

Prodromos Gerakios, Nikolaos Papaspyrou, and Konstantinos Sagonas. 2010. Race-Free and Memory-Safe Multithreading: Design and Implementation in Cyclone. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Madrid, Spain) *(TLDI '10)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/1708016.1708020

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 282–293. https://doi.org/10.1145/512529.512563

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems* 18, 2 (1996), 109–138.

Niels Hallenberg. 1996. A Region Profiler for a Standard ML compiler based on Region Inference. Student Project 96-5-7, Department of Computer Science, University of Copenhagen (DIKU).

Niels Hallenberg, Martin Elsman, and Mads Tofte. 2002. Combining Region Inference and Garbage Collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 141–152. https://doi.org/10.1145/512529.512547

Simon Helsen and Peter Thiemann. 2001. Syntactic Type Soundness for the Region Calculus. *Electronic Notes in Theoretical Computer Science* 41, 3 (2001), 1–19. https://doi.org/10.1016/S1571-0661(04)80870-3 HOOTS 2000, 4th International Workshop on Higher Order Operational Techniques in Semantics (Satellite to PLI 2000).

Fritz Henglein, Henning Makholm, and Henning Niss. 2001. A Direct Approach to Control-Flow Sensitive Region-Based Memory Management. *Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming* (07 2001). https://doi.org/10.1145/773184.773203

Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2004. Experience with Safe Manual Memory-Management in Cyclone. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/1029873.1029883

Mark P. Jones. 1994. A theory of qualified types. *Science of Computer Programming* 22, 3 (1994), 231–256. https://doi.org/10.1016/0167-6423(94)00005-0

Pierre Jouvelot and David Gifford. 1991. Algebraic Reconstruction of Types and Effects. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Orlando, Florida, USA) *(POPL '91)*. Association for Computing Machinery, New York, NY, USA, 303–310. https://doi.org/10.1145/99583.99623

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages. https://doi.org/10.1145/3158154

Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers - Type- and Scope-Safe Code Generation with Mutable Cells. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10017)*, Atsushi Igarashi (Ed.). 271–291. https://doi.org/10.1007/978-3-319-47958-3_15

Oleg Kiselyov and Chung-chieh Shan. 2008. Lightweight Monadic Regions. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Victoria, BC, Canada) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1411286.1411288

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press, USA.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science* 153 (06 2014). https://doi.org/10.4204/EPTCS.153.8

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

Matthew Lutze, Magnus Madsen, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2023. With or Without You: Programming with Effect Exclusion. *Proc. ACM Program. Lang.* 7, ICFP, Article 204 (aug 2023), 28 pages. https://doi.org/10.1145/3607846

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

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type inference with constrained types. *Theory and Practice of Object Systems* 5, 1 (1999), 35–55. https://doi.org/10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4

Lars Pareto. 2000. *Types for Crash Prevention.* Ph. D. Dissertation. Chalmers University of Technology, Gothenburg University, Gothenburg, Sweden.

François Pessaux and Xavier Leroy. 1999. Type-Based Analysis of Uncaught Exceptions. *ACM Transactions on Programming Languages and Systems* 22, 276–290. https://doi.org/10.1145/292540.292565

John Peterson and Mark Jones. 1993. Implementing Type Classes. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93).* Association for Computing Machinery, New York, NY, USA, 227–236. https://doi.org/10.1145/155090.155112

Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical Memory Management for Parallel Programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016).* Association for Computing Machinery, New York, NY, USA, 392–406. https://doi.org/10.1145/2951913.2951935

KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 113 (Aug. 2020), 30 pages. https://doi.org/10.1145/3408995

Martin Franz Sulzmann and Paul Hudak. 2000. *A General Framework for Hindley/Milner Type Systems with Constraints.* Ph. D. Dissertation. USA. AAI9973781.

Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006. Safe manual memory management in Cyclone. *Science of Computer Programming* 62, 2 (2006), 122–144. https://doi.org/10.1016/j.scico.2006.02.003 Special Issue: Five perspectives on modern memory management - Systems, hardware and theory.

Jean-Pierre Talpin and Pierre Jouvelot. 1994. The Type and Effect Discipline. *Inf. Comput.* 111, 2 (jun 1994), 245–296. https://doi.org/10.1006/inco.1994.1046

Mads Tofte and Lars Birkedal. 1998. A Region Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (jul 1998), 724–767. https://doi.org/10.1145/291891.291894

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).

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. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. 2022. *Programming with Regions in the MLKit (Revised for Version 4.7.2).* Technical Report. IT University of Copenhagen, Denmark.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14).* Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

David Walker, Karl Crary, and Greg Morrisett. 2000. Typed Memory Management via Static Capabilities. *ACM Trans. Program. Lang. Syst.* 22, 4 (jul 2000), 701–771. https://doi.org/10.1145/363911.363923

Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement Detection with Near-Zero Cost. *Proc. ACM Program. Lang.* 6, ICFP, Article 115 (aug 2022), 32 pages. https://doi.org/10.1145/3547646

Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2019. Disentanglement in Nested-Parallel Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 47 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371115

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (nov 1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Received 2023-07-11; accepted 2023-11-07