# Higher ranked region inference for compile-time garbage collection

Ivo Gabe de Wolff

Supervisors
dr. J. Hage
prof. dr. G. Keller

August 21, 2019

**Abstract**

In many languages, memory is managed by a garbage collector. Region based memory management forms an alternative and is used in the Rust and MLKit compilers. Most of the work is done ahead of time, by splitting the heap into regions, which have a lexical lifetime. Whereas Rust forces the programmer to write region annotations, they can be automatically inferred in MLKit using a type and effect system.

To improve the accuracy and reduce poisoning, we designed a higher ranked region analysis. We implemented this in the Helium Haskell compiler, such that we can analyse a real world language like Haskell with higher-rank precision. Furthermore, this will give us insights in the integration of regions with other optimization passes.

# Contents

# 1

# Introduction

In high level languages like Haskell [37], memory management is implicit. The programmer does not have to specify when memory can be released. Instead, the compiler and runtime must assure that unused objects will be released. This can be done with a garbage collector, which scans the heap for unreachable objects at runtime. Garbage collectors can be made very efficient. A much used argument is that the amortized cost of garbage collection tends to zero, when the available memory goes to infinity [6, page 206]. However, in practice we do not have infinite memory and the garbage collection will thus impact performance.

The interest in memory management comes partially from the memory wall. The performance of processors has increased a lot more than the speed of memory, which thus increases the relative cost of memory operations [57, 41]. This is very relevant to programs written in functional languages like Haskell, as they typically allocate a lot of objects and use many pointers. This may cause that the caches, which modern processors use to lower the cost of memory operations, perform poorly.

Stack allocation [17] can be more efficient than heap allocations [53, 42], as they have lower deallocation cost, provide better cache locality and cause less fragmentation than heap allocation. The limitation of the stack discipline is that values cannot 'escape': all values must have a lexical lifetime in the function which allocates them. *Region based memory management* [54] provides a solution to this, by essentially allowing a function to allocate objects in stack frames of a previous function call on the call stack. Region based memory management was researched in the MLKit compiler [53] and has lately been popularized in the Rust programming language [39].

The memory of a program is represented as a stack of regions. In a lambda calculus, regions can be implemented by adding two constructs [54]:

- `letregion` $\rho$ `in` $e$
- $e$ `at` $\rho$

The `letregion` construct allocates a region, evaluates the expression, deallocates the region and returns the value computed by the expression. The expression may thus use the region $\rho$ for intermediate values, but its return value should not be stored in that region. The regions have a lexical lifetime and thus form a stack of regions.

The `at` construct must be used for all expressions that directly allocate an object (such as closures and data types). It denotes that the object should be placed in the specified

region. The position of those region-constructs can automatically be inferred [52] for a well typed functional program.

The size of a region can be unbounded. Instead of allocating the regions directly on the stack, we will allocate the regions on the heap and put pointers to the regions on the stack. Regions on the heap are represented as a linked list of blocks, such that they can grow when needed. As some of the advantages were lost, this approach gave large overhead. Tofte et al. [52] did however notice that most of the regions contained only one value.

To reduce the overhead, a multiplicity analysis was developed [55], which gives an upperbound on the number of objects that will be placed in a region. Finite regions can now be allocated on the stack. Based on experiments, more than 90% of the allocations were done on the stack for many programs.

## 1.1 Memory leaks

Region based memory management can cause unbounded memory leaks. Situations exists where many objects, which will not be used, cannot be released. The same applies to garbage collection, or in fact any implementation of automatic memory management: the question whether some object will be used later on is undecidable, as a consequence of the halting problem. Region based memory management and garbage collection are thus both an approximation, and they are incomparable: some memory can be reclaimed by garbage collection which cannot be reclaimed by a region implementation, and the other way around [21].

A garbage collector treats all reachable objects as live, whereas reachability does not mean that it will be used later on. If a program constructs a list and the list will only be used to compute its length later on, then the elements of the list may be reclaimed. A garbage collector cannot do that, whereas those values may be put in a separate region with region inference and can thus be released before the list is released.

For the other way around, consider the following Haskell function. This example can cause a memory leak with region based memory management.

```
f :: [Int] -> [Int]
f xs = [1, 2] ++ drop 2 xs
```

Function $f$ removes the first two elements of the list and append values 1 and 2. Lists are represented as linked lists in Haskell. All list nodes are required to be in the same region. Consider that function $f$ is applied $N + 1$ times (on its own output). This will cause that $2N$ nodes are in the same region as the output list, but those nodes are not reachable and thus form a memory leak.

## 1.2 Runtime

The required runtime for region based memory management consists of three functionalities [9]. First, we must be able to construct a new region. For finite regions, we allocate a block on the stack and for infinite regions we allocate a block on the heap.

Second, we must perform allocations of objects in regions. We do so by keeping a pointer in a block, pointing at the first free position. This pointer is increased after each allocation. When a block is full, which may only happen for infinite regions, we allocate a new block on the heap, which is linked to the previous block as a linked list.

Third, we must be able to release a region and implicitly deallocate all objects allocated in the region. For finite regions allocated on the stack, we move the stack pointer back. For infinite regions, we put the first block of the linked list on a free list. Later regions

will reuse those blocks from the free list. Deallocations can thus happen in constant time, independent of the number of objects in the region [9].

## 1.3 Type and effect system

Program analyses can be written as a type and effect system [43], which works by annotating the underlying type system of a language with additional information. This may be information on either the value (for instance the sign of a number) or the computation of that value (for instance, resource usage). The analysis itself can then be written by adapting an existing type inference algorithm.

We implemented the higher ranked region analysis in Helium, which uses a type system based on System F$\omega$, which is an extension of System F [20] with data types.

$$\begin{array}{rll} \tau ::= & (\rightarrow) & \textit{Function type} \\ \mid & \tau\ \tau & \textit{Type application} \\ \mid & \forall\alpha.\ \tau & \textit{Quantified type} \\ \mid & \alpha & \textit{Type variable} \\ \mid & !\tau & \textit{Strict type} \\ \mid & D & \textit{Data type} \\ \mid & (\ ,\ )\ \mid\ (\ ,,\ )\ \mid\ \ldots & \textit{Tuple of arity } n > 1 \\ \mid & () & \textit{Unit type (tuple of arity 0)} \end{array} \tag{1.1}$$

We write $\tau_1 \rightarrow \tau_2$ for $(\rightarrow)\ \tau_1\ \tau_2$ and $(\tau_1, \tau_2)$ for $(,)\ \tau_1\ \tau_2$. The type $\tau$ in a strict type $!\tau$ must have kind $*$ and cannot be a strict type; we do not allow the type $!!\tau$. We do not make the kinds of type variables explicit, as we do not need this information in the analysis.

We write $\tau_1[\alpha := \tau_2]$ for the substitution of type variable $\alpha$ with $\tau_2$ in type $\tau_1$. We add the normalization rule $(\forall\alpha.\ \tau_1)\tau_2 = \tau_1[\alpha := \tau_2]$ to the type system. We will not consider name collisions in this thesis.

The type system supports tuples of all arities larger than 1, but we will usually only give details on tuples of arity 2 in this thesis, as larger tuples behave similarly.

The arity of a type is the number of value arguments that it accepts, ignoring type arguments.

$$\begin{aligned} \mathit{arity}(\tau_1 \rightarrow \tau_2) &= 1 + \mathit{arity}(\tau_2) \\ \mathit{arity}(\forall\alpha.\ \tau) &= \mathit{arity}(\tau) \\ \mathit{arity}(\tau) &= 0 \text{ if } \tau \text{ is not a function type or a quantification} \end{aligned} \tag{1.2}$$

## 1.4 Higher ranked analyses

Higher ranked type inference, which allows polymorphic functions as arguments, is undecidable [45] and thus require annotations from the programmer. For instance, consider a function which takes the identity function as its argument. An explicit type annotation is needed when writing such function in Haskell.

```
f :: (forall a. a -> a) -> Int -> Bool -> (Int, Bool)
f id x y = (id x, id y)
```

However, as analyses are usually less expressive than a type system, they can be made higher ranked depending on the analysis [51, 30, 34].

Different calls to an argument of a function will not interfere with each other. In practice this will mean for region inference that a function `\f x y -> let u = f x in let v = f y in (u, v)` will not enforce that `x` and `y` are stored in the same region, nor that `u` and `v` are in the same region.

The main differences between our analysis and an ordinary type and effect systems are the following:

- We assume that the program is already typed. The analysis thus does not have to infer the types, it only has to fill in the annotations
- We do not have annotations on the left hand side of a function. Normally one would have annotations there. Instead, the annotation on a function is actually a function taking the annotations of its argument.

## 1.5 Research questions

**Research question 1.** *Can region inference be made higher ranked?*

Other program analyses have been made higher ranked, which made them more precise for higher order functional programs.

**Research question 2.** *Can the existing work on higher ranked type and effects systems be adapted to a real programming language like Haskell?*

Previously, higher ranked analyses were implemented on a toy language, without polymorphism and data types. Can the same methods be used, when extending such language with polymorphism and data types?

**Research question 3.** *Can the region based memory management be integrated in the Helium compiler?*

The Helium compiler performs optimizations such as tail recursion, strictness analysis and thunks with multiple arguments. What are the implications of those optimizations on the region analysis and runtime?

## 1.6 Contributions

Compared to the work by Tofte e.a. [53], we designed a different region inference algorithm. This algorithm is higher ranked and adapted to Haskell and thus to lazy evaluation.

We extended the existing work on higher ranked analyses [51, 34, 30] in different ways. We modified the lambda calculus for annotations as used in [51] to handle polymorphism in the source language. We propose a different solution to the lack of a canonical form of lambda functions, which complicates the computation of fixpoints. However, this approach will need to be worked out further in future work, as it is not yet general enough for all programs. Finally we applied the techniques to region inference, which required making the annotation language even larger.

The analysis was implemented in Helium, a compiler for Haskell developed at Utrecht University. Due to time limitations, the code generation of the regions was not implemented. The implementation has two known bugs. For some mutual recursive functions, the assignment of additional region arguments is wrong. The containment constraints for data types may cause cyclic constraints, as described in chapter 9.1.1. Furthermore, we did not have enough time to implement annotations on data types, but we discuss the design for those in chapter 11.1.3. The implementation was implemented in the

branch `codegen-llvm-regions` of Helium, which can be found at `https://github.com/Helium4Haskell/helium/tree/codegen-llvm-regions`.

Besides the region inference analysis itself, various other changes to the compiler were needed. The largest of them was to make the intermediate languages of Helium, Core and Iridium, typed. As this was a rather large change, we haven't made the code generation for `deriving` typed. We modified the thunk evaluation scheme, as the old scheme made it hard to reason about programs statically. Furthermore, we implemented tail call optimization, which converts tail recursive function to loops in Iridium. We added a simple strictness analysis, which only operates on the first order parts of a program. It does support strict fields. These two passes were implemented as they have a large influence on the region inference.

# 2

# Analysis overview

The goal of the region inference analysis is to decide where regions are allocated and, for every object, in which region it will be allocated. As objects may have nested fields, we may assign multiple regions to an object such that its fields are allocated in different regions. We do so by assigning multiple region variables to a value, depending on its type. This is formalized in chapter 3.

The analysis infers constraints on those regions. Those constraints are contained in annotations. To handle higher order functions, these annotations are functions taking both the annotations and regions of the argument.

The analysis will try to unify regions, to reduce runtime overhead and improve accuracy. Furthermore, it analyses which regions do not escape out of a function, as we then can decide to allocate the region within that function.

## 2.1 Structure

The remainder of this thesis is organized as follows. We will introduce the general region inference, by introducing region variables in chapter 3, the types of annotations in chapter 4 and the annotation language in chapter 5. We then consider the backend of Helium in chapter 6, to give some context on the implementation of the region inference analysis. We present the implementation of the region inference analysis in chapter 7 and discuss the interaction with other compiler optimizations present in the Helium compiler in chapter 8. In chapter 9, we give the details on the graph-based algorithms which we use in the analysis.

# 3

# Region arguments

Region inference decides for each object, in which region it will be allocated. We reference those regions using region variables, which are denoted by $\rho$[1]. Nested fields of an object may be allocated in different regions.

Each value is annotated with region arguments. As nested fields may be placed in different regions, we may assign multiple region arguments. The number of region arguments of a value depends on its type. We need different regions for a value as thunk and as an evaluated value. We may also need regions for nested fields, for instance the elements of a tuple or a list. We write $\hat{\rho}$ to denote a tree-structure of region arguments and use tuples of regions to structure multiple region arguments. As we will see later, the tree structure of tuples is necessary to handle polymorphism.

$$\begin{aligned} \hat{\rho} ::= {} & \rho \\ | {} & (\hat{\rho}, \hat{\rho}, \ \dots) \end{aligned} \tag{3.1}$$

Note that the tuple may also be empty (a unit), which we use for types which do not require additional region arguments, such as integers. We have two special regions, the global region $\rho_{global}$ and the bottom region $\rho_{\perp}$. The lifetime of the global region is the full execution of the program. This region is used for global variables and for code that cannot be analysed, for instance because complex data types are used. The bottom region has no meaning at runtime. All regions outlive the bottom region, but no other region is outlived by the bottom region. It is used in the analysis for regions which do not escape out of a function.

We call the "type" of the region arguments the *region sort*, denoted by $\hat{P}$. A region sort is either a monomorphic sort, denoting a single region, a (possibly empty) tuple of region sorts or a polymorphic region sort.

$$\begin{aligned} \hat{P} ::= {} & P && \textit{Monomorphic region sort} \\ | {} & P\langle\ \alpha\ \overline{\tau}\ \rangle && \textit{Polymorphic region sort} \\ | {} & \left(\hat{P}, \hat{P}, \ \dots\right) && \textit{Tuple} \end{aligned} \tag{3.2}$$

[1] $P$ and $\rho$ are the Greek letter rho.

The tuples of region sorts allow us to create a tree of region arguments. A leaf of that tree may be a polymorphic region sort, which is used for values of a polymorphic type. When instantiating such polymorphic type, the polymorphic region sorts will be replaced by the region sorts of the instantiated type. That is, we replace the node of the polymorphic region sort by a subtree of the instantiated type.

In many examples, we use integers, which would normally not be heap allocated. This is because they fit in a word and have the same size as a pointer. However, in the examples we assume that integers are heap allocated for brevity. In an implementation, one may just forget the region arguments of integers after the analysis, such that no special case is needed for values which are not heap allocated.

We write $\hat{\rho} : \hat{P}$ to denote that regions $\hat{\rho}$ have region sort $\hat{P}$. For clarity, we will sometimes interleave both tree structures and for instance write $(\rho_1 : P,\ \rho_2 : P)$ instead of $(\rho_1, \rho_2) : (P, P)$.

## 3.1 Region assignment

We define a mapping $P_\Gamma : \tau \to \hat{P}$ from types of kind $*$ to region sorts. For strict types, we need one region to store the value and for other types we need one region for the thunk and one for the evaluated value. Depending on the type, we may also need regions for nested values, for instance the elements of a thunk, which are found in $P^\circ_\Gamma$. The subscript $\Gamma$ represents the effect environment, which contains information on the data types of the program. We will later extend the effect environment with the variables which are in scope.

$$\begin{aligned} P_\Gamma(!t) &= (P, P^\circ_\Gamma(t)) \\ P_\Gamma(t) &= (P, P, P^\circ_\Gamma(t)) \text{ if } t \text{ is not strict} \end{aligned} \tag{3.3}$$

Function $P^\circ_\Gamma$ gives the region sort for the fields of a type. For types which do not have fields, like integers or unit, we thus get an empty tuple. For functions we also do not have additional region parameters, as we will later specify the regions of the arguments and return value in the annotation on the function. For each element of a tuple, we need a region to store the thunk for that element, a region for the evaluated value and a region for fields of the element.

The region sort of data types is found in the effect environment $\Gamma$. We will later discuss how region arguments are assigned to data types.

$$\begin{aligned} P^\circ_\Gamma(\tau_1 \to \tau_2) &= () \\ P^\circ_\Gamma(()) &= () \\ P^\circ_\Gamma((\tau_1, \tau_2)) &= (P, P, P^\circ_\Gamma(\tau_1), P, P, P^\circ_\Gamma(\tau_2)) \\ P^\circ_\Gamma(D\ \tau_1 \dots \tau_n) &= \hat{P} \text{ if the region sort of data type } D \text{, instantiated with} \\ &\qquad t_1 \dots t_n \text{, in environment } \Gamma \text{ is } \hat{P}. \\ P^\circ_\Gamma(\alpha\ \tau_1 \dots \tau_n) &= P\langle \alpha\ \tau_1 \dots \tau_n \rangle \end{aligned} \tag{3.4}$$

## 3.2 Instantiation

Region sorts may be instantiated, which means that a type variable is substituted with some type. If we instantiate the type variable of a polymorphic region sort $(P\langle\ \alpha\ \overline{\tau}\ \rangle)$, we expand it to the region sort of the instantiated type. A substitution of type variable

$\alpha$ with type $\tau$ in region sort $\hat{P}$ with effect environment $\Gamma$ is written as $\hat{P}[\alpha := \tau]_\Gamma$. We define the substitution using the substitution on types, written as $\tau'[\alpha := \tau]$.

$$
\begin{aligned}
P[\alpha := \tau]_\Gamma &= P \\
P\langle \alpha'\ \tau_1\ \ldots\ \tau_n\rangle[\alpha := \tau]_\Gamma &= \begin{cases} P^\circ_\Gamma(\tau\ \tau_1[\alpha := \tau]\ \ldots\ \tau_n[\alpha := \tau]) & \text{if } \alpha = \alpha' \\ P\langle \alpha'\ \tau_1[\alpha := \tau]\ \ldots\ \tau_n[\alpha := \tau]\rangle & \text{otherwise} \end{cases} \\
\left(\hat{P}_1, \ldots \hat{P}_n\right)[\alpha := \tau]_\Gamma &= \left(\hat{P}_1[\alpha := \tau]_\Gamma,\ \ldots\ \hat{P}_n[\alpha := \tau]_\Gamma\right)
\end{aligned}
\tag{3.5}
$$

The monomorphic region sort does not change in a substitution. For a polymorphic region sort, we must apply the substitution to the type arguments, which are denoted by $\tau_1\ \ldots\ \tau_n$ in equation 3.5. Furthermore, we must check whether the type variable $\alpha'$ in the polymorphic region sort is being instantiated. If that is the case, we must replace the polymorphic region sort with the sort of the instantiated type, which function $P^c_\Gamma irc$ yields. Otherwise, we still leave it as a polymorphic region sort. For tuples, we propagate the substitution to its elements.

## 3.3 Lifetime relation

The analysis computes a relation on the lifetimes of region variables in a program. We express this relation using $\geq$, which means "lives at least as long as". For brevity we will however say "$\rho_1$ outlives $\rho_2$" instead of "$\rho_1$ lives at least as long as $\rho_2$". The relation is reflexive and transitive, which is also known as a preorder [48] and can be seen as a partial order modulo unifications.

Given a relation $\mathcal{R}$, we write $x \leq_\mathcal{R} y$ for $(x, y) \in \mathcal{R}$, $x \geq_\mathcal{R} y$ for $(y, x) \in \mathcal{R}$ and $x \equiv_\mathcal{R} y$ for $(x, y) \in \mathcal{R}\ \wedge\ (y, x) \in \mathcal{R}$.

We have two special regions, $\rho_{global}$ and $\rho_\bot$. The global region, $\rho_{global}$ has a lifetime of the full lifetime of the program. This is used for global variables, which should not be deallocated during the execution of the program. The global region must be the top in a lifetime relation $\mathcal{R}$. That is, for all $\rho$, $\rho_{global} \geq_\mathcal{R} \rho$. The bottom region, $\rho_\bot$, must be the bottom element of the relation, thus for all $\rho$, $\rho \geq_\mathcal{R} \rho_\bot$.

**Definition 3.3.1** (Reflexive relation)**.** *A relation $\mathcal{R} \subset S \times S$ on set $S$ is reflexive if for all $x \in S$, $(x, x) \in \mathcal{R}$ holds [48].*

**Definition 3.3.2** (Transitive relation)**.** *A relation $\mathcal{R} \subset S \times S$ on set $S$ is transitive if $(x, y) \in \mathcal{R}\ \wedge\ (y, z) \in \mathcal{R}$ implies $(x, z) \in \mathcal{R}$ for all $x, y, z \in S$ holds [48].*

**Definition 3.3.3** (Lifetime relation)**.** *A lifetime relation $\mathcal{R} \subset P \times P$ is a reflexive, transitive relation, such that for all $\rho$, $\rho_{global} \geq_\mathcal{R} \rho$ and $\rho \geq_\mathcal{R} \rho_\bot$. We write $R$ for the set of all lifetime relations.*

We write a lifetime relation $\mathcal{R}$ as $[\![a \geq b, c \geq d, \ldots]\!]$, where all pairs $\rho_1, \rho_2$ should be present as $\rho_1 \geq \rho_2$ between the brackets, for which $\rho_1 \geq_\mathcal{R} \rho_2$ and $\rho_1 \neq \rho_2$, $\rho_1 \neq \rho_{global}$ and $\rho_2 \neq \rho_\bot$. That is, we specify all outlive-relations, except for reflexivity, $\rho_{global}$ as top element and $\rho_\bot$ as bottom, as we leave those implicit.

**Definition 3.3.4** ($\sqsupseteq$, partial order on lifetime relations)**.** *We say that a lifetime relation $\mathcal{R}_1$ is at least as precise (or constrained) as $\mathcal{R}_2$ and write $\mathcal{R}_1 \sqsupseteq \mathcal{R}_2$ if for all $\rho_1$, $\rho_2$, $\rho_1 \geq_{\mathcal{R}_2} \rho_2$ implies $\rho_1 \geq_{\mathcal{R}_1} \rho_2$.*

From the definition of $\sqsupseteq$ it follows directly that $\mathcal{R}_1 \sqsupseteq \mathcal{R}_2$ is equivalent to $\mathcal{R}_1 \supseteq \mathcal{R}_2$. Thus, $\sqsupseteq$ is a partial order.

We write $[\![a \geq b, c \geq d, \ldots]\!]^*$ for the smallest lifetime relation satisfying the given constraints, namely the transitive closure [48] of these constraints with $\rho_{global}$ added as top element.

**Definition 3.3.5** ($\sqcup$, join of lifetime relations)**.** *We define the join of lifetime relations $\mathcal{R}_1$ and $\mathcal{R}_2$, denoted by $\mathcal{R}_1 \sqcup \mathcal{R}_2$, as the smallest lifetime relation $\mathcal{R}$ such that $\mathcal{R} \sqsupseteq \mathcal{R}_1$ and $\mathcal{R} \sqsupseteq \mathcal{R}_2$.*

Finally, we write $\bot$ for an empty relation.

## 3.4 Region sort of data types

The region sorts of data types are constructed in the following way. First we start by finding the binding groups of data types. Each binding group contains one or more data types, such that there is no mutual recursion between binding groups, ignoring recursion in a function type. The dependencies between binding groups thus form a directed acyclic graph. We process the binding groups in a topological order. This is similar to how binding groups of functions are dealt with in the type inferencer of Helium [23].

All data types in a binding group will get the same region sort. We write temporarily to the effect environment that all these data types have zero region arguments. With this environment we compute the sort regions of all fields of all constructors in the binding group. These sorts are concatenated and flattened to a single list, which becomes the region sort of all data types in the binding group.

Finally we construct a mapping between the resulting list and the fields of each constructor. For recursive positions, we instantiate the regions with the same region arguments.

## 3.5 Containment

The containment property means intuitively that the fields of a value outlive the value itself. For instance for a tuple, the elements should live at least as long as the tuple. We formalize this with a containment function $\mathcal{C}_\Gamma : \tau \to \hat{P} \to R$, taking the type and regions of a value and returning a constraint between those. The type should be of kind $*$ and the region arguments should have sort $P_\Gamma(\tau)$.

$$\begin{aligned} \mathcal{C}_\Gamma(!\tau, (\rho_2, \hat{\rho}_3)) &= \mathcal{C}^\circ_\Gamma(\tau, \rho_2, \hat{\rho}_3) \\ \mathcal{C}_\Gamma(\tau, (\rho_1, \rho_2, \hat{\rho}_3)) &= [\![\rho_2 \geq \rho_1]\!] \sqcup \mathcal{C}^\circ_\Gamma(\tau, \rho_2, \hat{\rho}_3) \end{aligned} \tag{3.6}$$

For non-strict values, we require that the region in which the value is stored outlives the region containing the thunk. This prevents that an evaluated thunk points at a deallocated value.

Note that $\hat{\rho}_3$ may not just be a single region, but may represent an empty list (for types without additional region arguments) or a list of region arguments. The constraints between these regions and the region of the value ($\rho_2$) are specified in $\mathcal{C}^\circ_\Gamma : \tau \times P \times \hat{P} \to R$. First we consider tuples. A tuple has three region arguments for each field, namely a region to store the thunk, one for the value and one for nested fields of the element. The last of these can itself be a list of multiple region arguments. We require that the region containing the thunk outlives the region of the tuple itself and that the region of the value outlives the thunk. Implicitly this means that the region of the element value must outlive the region of the tuple. Furthermore, the element in the thunk may have more region arguments in $\rho_3$, which can give more constraints.

$$\begin{aligned}
&\mathcal{C}^{\circ}_{\Gamma}((\tau_1, \tau_2), \rho, (\rho_{1t}, \rho_{1v}, \hat{\rho}_1, \rho_{2t}, \rho_{2v}, \hat{\rho}_2)) \\
&\quad = [\![\rho_{1v} \geq \rho_{1t}, \rho_{1t} \geq \rho, \rho_{2v} \geq \rho_{2t}, \rho_{2t} \geq \rho]\!]^* \\
&\qquad \sqcup \mathcal{C}^{\circ}_{\Gamma}(\tau_1, \rho_{1v}, \hat{\rho}_1) \\
&\qquad \sqcup \mathcal{C}^{\circ}_{\Gamma}(\tau_2, \rho_{2v}, \hat{\rho}_2)
\end{aligned} \tag{3.7}$$

For polymorphic values, we have one polymorphic region variable assigned for the fields of that type. We require that that region outlives the region of the evaluated value. Note that this region argument may later be replaced by multiple region variables, when the type variable of this polymorphic region variable is instantiated.

$$\mathcal{C}^{\circ}_{\Gamma}(\alpha\ \tau_1 \ldots \tau_2, \rho, \rho_1) = [\![\rho_1 \geq \rho]\!] \tag{3.8}$$

Finally, we have no further containment rules for types without region arguments for nested fields.

$$\mathcal{C}^{\circ}_{\Gamma}(\tau, \rho, ()) = \bot \tag{3.9}$$

### 3.5.1 Containment for data types

We require for data types that all nested region variables outlive the region of the value. Furthermore, we require all containment relations of the fields of the constructors. The latter is only needed for consistency for the garbage collector. These constraints would still be generated when pattern matching or constructing objects, but as a garbage collector must always be able to follow pointers, we must add those additional containment constraints. This has the consequence that containment constraints may become cyclic. If that occurs, we must unify these region arguments of the data type, as we would run into problems later if we generate cyclic outlive constraints. An example of this is shown in chapter 9.1.1.

# 4

# Annotation sorts

Besides region arguments, we will also assign annotation arguments to values. Whereas region arguments are concrete in the sense that they directly represent the region in which a value is stored at runtime, annotation arguments are more abstract as they describe the relations between region variables. To be more precise, annotations describe relations between the regions of the arguments and the return value of a function.

In this chapter, we introduce the types of annotations. We call those types *sorts*.

$$
\begin{array}{lr}
s ::= R & \textit{Relation} \\
\quad \mid (\ \overline{s}\ ) & \textit{Tuple} \\
\quad \mid \left[s; \hat{P}\right]_l \to s & \textit{Function} \\
\quad \mid \forall \alpha.\ s & \textit{Quantification} \\
\quad \mid \Psi\langle\ \alpha\ \overline{\tau}\ \rangle & \textit{Polymorphic sort}
\end{array}
\tag{4.1}
$$

$$
l ::= \perp\!\!\!\uparrow \mid \emptyset
$$

A relation, $R$, is a lifetime relation on regions. A tuple sort is used for types with multiple annotations, similar to the tuple used in region sorts.

The function sort is used to retrieve the regions and annotations of an argument or return value of a function (in the source language). We may limit how region arguments are used in a function. We annotate region arguments with a lifetime context, denoted by $l$. The lifetime context $\perp\!\!\!\uparrow$, pronounced *local bottom*, denotes that those argument may only be used on the right hand side of an outlives constraint ($\geq$). Thus, the function may not give constraints on the lifetime of those region variables, but these region variable can be used to give constraints on other region variables. When we do not annotate the region argument with such arrow, it may be used on both sides of the outlives constraints. We write $l$ for the lifetime context, $l ::= \ \perp\!\!\!\uparrow \mid \emptyset$. This lifetime context only applies to the region arguments of an annotation function.

The quantification sort is used for polymorphic values. Note that this quantifies over types instead of sorts. Similar to region sorts, we also have a polymorphic sort here. This sort will be replaced when its type variable is instantiated.

## 4.1 Instantiation

Quantified sorts can be instantiated with a type. When doing so, we substitute all occurrences of the type variable with the instantiated type. If the type variable is used in a polymorphic region ($P\langle\alpha\ \overline{\tau}\rangle$) or annotation sort ($\Psi\langle\alpha\ \overline{\tau}\rangle$), we will substitute that sort with the regions or annotations of the instantiated type.

A substitution of type variable $\alpha$ with type $\tau$ in sort $s$ in effect environment $\Gamma$ is denoted by $s[\alpha := \tau]_\Gamma$. For brevity we won't consider name collisions. We define the substitution using the substitutions on types ($\tau'[\alpha := \tau]$) and on region sorts ($\hat{P}[\alpha := \tau]_\Gamma$).

$$
\begin{aligned}
R[\alpha := \tau]_\Gamma &= R \\
(s_1,\ \ldots\ s_n)\,[\alpha := \tau]_\Gamma &= (s_1[\alpha := \tau]_\Gamma,\ \ldots\ s_n[\alpha := \tau]_\Gamma) \\
\left(\left[s_1; \hat{P}\right]_l \to s_2\right)[\alpha := \tau]_\Gamma &= \left[s_1[\alpha := \tau]_\Gamma; \hat{P}[\alpha := \tau]_\Gamma\right]_l \to s_2[\alpha := \tau]_\Gamma \\
(\forall\alpha'.\ s)\,[\alpha := \tau]_\Gamma &= \forall\alpha'.\ s[\alpha := \tau]_\Gamma \\
\Psi\langle\alpha'\ \tau_1\ \ldots\ \tau_2\rangle[\alpha := \tau]_\Gamma &= \begin{cases} \Psi_\Gamma(\tau\ \tau_1[\alpha := \tau]\ \ldots\ \tau_n[\alpha := \tau]) & \text{if } \alpha = \alpha' \\ \Psi\langle\alpha'\ \tau_1[\alpha := \tau]\ \ldots\ \tau_n[\alpha := \tau]\rangle & \text{otherwise} \end{cases}
\end{aligned}
\tag{4.2}
$$

## 4.2 Annotation assignment

Similarly to region sorts, we define a mapping $\Psi_\Gamma : \tau \to s$ from types to sorts. We annotate a function type $\tau_1 \to \tau_2$ with annotations which describe the constraints on the regions of the return value. The assigned annotation has a function sort, taking the annotations and regions of $\tau_1$ and returning a tuple of two elements. The first element has sort $[(), P] \to [(), P_\Gamma(!\tau_2)]_{\uparrow\perp} \to R$ and is a function taking the region of the previous thunk, used for partial applications as described below, and the region variables of the return value. The function returns the constraints on those return regions in terms of the other regions. The second element of the tuple contains the annotations of $\tau_2$.

$$
\Psi_\Gamma(\tau_1 \to \tau_2) = [\Psi_\Gamma(\tau_1); P_\Gamma(\tau_1)] \to ([(), P] \to [(), P_\Gamma(!\tau_2)]_{\uparrow\perp} \to R,\ \Psi_\Gamma(\tau_2)) \tag{4.3}
$$

For other types, we propagate the annotations on nested functions. For instance, for a tuple type, we create a tuple with the annotations of the elements.

$$
\begin{aligned}
\Psi_\Gamma(Int) &= () \\
\Psi_\Gamma(()) &= () \\
\Psi_\Gamma(!\tau) &= \Psi_\Gamma(\tau) \\
\Psi_\Gamma((\tau_0, \tau_1)) &= (\Psi_\Gamma(\tau_0), \Psi_\Gamma(\tau_1)) \\
\Psi_\Gamma(D\ \tau_1 \ldots \tau_n) &= s \text{ if the annotation argument sort of data type } D, \\
&\qquad \text{instantiated with } \tau_1 \ldots \tau_n, \text{ in environment } \Gamma \text{ is } s. \\
\Psi_\Gamma(\alpha\ \tau_1 \ldots \tau_n) &= \Psi\langle\ \alpha\ \tau_1 \ldots \tau_n\ \rangle
\end{aligned}
\tag{4.4}
$$

We thus assign both region and annotation variables, which we will clarity with an example. Consider a value of type `Int -> Bool`. This value gets two region arguments, namely one for the function if it is not yet evaluated to weak head normal form and one where the region is stored after evaluating to weak head normal form. These regions thus have a direct meaning at runtime. Furthermore we assign an annotation variable, which describes the relation between the argument of the function, of type `Int`, and the result, of type `Bar`.

### 4.2.1 Partial applications

When partially applying a function, a thunk object is constructed containing a pointer to the function and its arguments. Thunks may form a linked list, which happens when calling a partially applied function, whose arity is at least two. As an example, consider the following code:

```
f x y z = x + y + z

a = f 1
b = a 2
```

Variable `b` contains a thunk object, pointing at `a`. We thus need a containment constraint, saying that the region of thunk `a` outlives the region of thunk `b`. The annotation function thus needs to get the region of the previous thunk as an argument. We give this region variable in an additional lambda, $[(), P]$, as seen in the sort of a function:

$$\Psi_\Gamma(\tau_1 \to \tau_2) = [\Psi_\Gamma(\tau_1); P_\Gamma(\tau_1)] \to ([(), P] \to [(), P_\Gamma(!\tau_2)]_{\uparrow\!\perp} \to R,\ \Psi_\Gamma(\tau_2))$$

Note that this argument is only used for partial applications. When partially applying a function with only the first argument, as we do in the definition of `a`, we use this region as well. In this case, the thunk will not point at a previous thunk. Instead it points at a function, whose lifetime is the full execution time of the program. We thus instantiate this region argument with the global region, $\rho_{global}$. This will not give additional constraints, as the global region always outlives all regions.

# 5

# Annotation language

The analysis works with annotations from a typed lambda calculus, typed with the sorts introduced in the previous chapter. We introduce the annotation language in this chapter. We start by introducing the syntax and give some insights on the design of the language with regards to the features of the source language, Haskell. We discuss the sorting rules (the type system of the annotation language) and the evaluation rules.

The syntax of the annotation language is shown in figure 5.1. The simplest annotation is bottom, $\bot$, which means that there are no constraints on the argument and return regions of some function.

A lambda or an abstraction is written as $\lambda \left[\psi : s; \hat{\rho} : \hat{P}\right]_l \mapsto a$. It introduces both region and annotation variables. It is used to accept the annotations and regions of arguments and the return value of a Haskell function. Introduced annotation variables can of course be used in the body of a lambda and are denoted by a $\psi$. The abstraction is

$$
\begin{array}{lr}
a ::= \bot & \textit{Bottom} \\
\mid \lambda \left[\psi : s; \hat{\rho} : \hat{P}\right]_l \mapsto a & \textit{Abstraction} \\
\mid \psi & \textit{Variable} \\
\mid a\ [a; \hat{\rho}]_l & \textit{Application} \\
\mid [\![\ \overline{\rho \geq \rho}\ ]\!] & \textit{Relation} \\
\mid \forall \alpha.\ a & \textit{Quantification} \\
\mid a\ \{\tau\} & \textit{Instantiation} \\
\mid (\ \overline{a}\ ) & \textit{Tuple} \\
\mid \pi_n(a) & \textit{Projection} \\
\mid a\ \sqcup\ a & \textit{Join} \\
\mid fix :\ s.\ a & \textit{Least fixpoint} \\
\mid fix\ escape \left[n; \hat{P}\right] : s.\ a & \textit{Least fixpoints with escape check}
\end{array}
\tag{5.1}
$$

**Figure 5.1:** Language of annotations

annotated by a lifetime context, which denotes on which sides of the outlives constraint ($\geq$) the region variables may be used, the same as the lifetime context on a function sort.

Lambdas can be applied with an application, written as $a\ [a; \hat{\rho}]_l$. The application provides both an annotation and region variables.

A relation, written as a list of outlive constraints within brackets, describes a relation on the lifetimes of region variables. The constraints in the list must be unique and transitively closed. Reflexivity and constraints of the form $\rho_{global} \geq \rho$ or $\rho \geq \rho_\perp$ are implicit and should not be listed.

To accommodate polymorphism, we need quantification ($\forall\alpha.\ a$) and instantiation $a\ \{\tau\}$. Instantiating an annotation will cause that polymorphic region sorts and polymorphic annotation sorts of the instantiated type variable are instantiated with the given type.

The language has tuples, as some types need multiple annotations. Tuples are represented as a comma-separated list of annotations. To extract elements out of a tuple, we have (zero-based) projection, $\pi_n(a)$.

Furthermore, we have a join operator, which evaluates to the least annotation (with respect to the partial order which we will define in chapter 3.3) greater than both of its arguments.

To analyse recursive programs, we introduce two kinds of least fixpoint combinators. The first fixpoint combinator, $fix :\ s.\ a$, evaluates to the least annotation $a'$ of sort $s$ such that $a\ a' = a'$. This combinator is used to analyse functions with imperative loops, as we applied the analysis to Iridium, a language with imperative control flow. The second fixpoint combinator, $fix\ escape[n; s] : s.\ a$, is used for (mutual) recursive functions. Besides computing the least fixpoint if $a$, it will also try to remove region arguments from $a$. The removed regions may either be allocated within the analysed function as they do not escape, or they may be unified with some other region variables.

The language features both region and annotation variables. Annotation variables, denoted by $\psi$, are the ordinary variables that one expects in a lambda calculus. They may be applied, instantiated or used as argument in an application. Region variables, denoted by $\rho$, come from the domain of our analysis. They are meant to be used in lifetime relations. To handle function calls in the source language, they may also be passed in application. An application in our annotation language namely passes both region arguments and annotation arguments.

## 5.1 Partial order on annotations

In chapter 3.3, we defined a partial order on lifetime relations, e.g. annotations of sort $R$. We extend the definition to annotations of arbitrary sorts, inductively over the sorts.

**Definition 5.1.1** (Partial order on annotations)**.** *Let $a$ and $a'$ be annotations of sort $s$ without free type and annotation variables. Annotation $a$ is at least as precise as $a'$, denoted by $a \sqsubseteq a'$, if and only if one of the following holds:*

**Lifetime relation** *Case $s = R$. The arrow $\longrightarrow$ denotes the "evaluates to" relation, which we will introduce below. Annotation $a$ is at least as precise as $a'$ if $a \longrightarrow a_1$, $a' \longrightarrow a'_1$ and $a_1 \sqsubseteq a'_1$ with respect to the partial order on lifetime relations from chapter 3.3.*

**Function** *Case $s = [s_1; \hat{P}] \to s_2$. If for all annotations $a_1$ of sort $s$ and regions $\hat{\rho}$ of region sort $\hat{P}$, $a[a_1; \hat{\rho}] \sqsubseteq a'[a_1; \hat{\rho}]$, then $a \sqsubseteq a'$.*

**Quantification** *Case $s = \forall\alpha.\ s'$. If for all types $\tau$, $a\ \{\tau\} \sqsubseteq a'\ \{\tau\}$ then $a \sqsubseteq a'$.*

**Tuple** *Case* $s = (s_1, \ldots s_n)$. *If for all indices* $1 \leq i \leq n$ *it holds that* $\pi_i(a) \sqsubseteq \pi_i(a')$, *then* $a \sqsubseteq a'$.

We write $a = a'$ when $a \sqsubseteq a'$ and $a' \sqsubseteq a$. Note that we do not define the relation on annotations of a polymorphic sort $(\Psi\langle\alpha\ \tau_1\ \ldots\ \tau_n\rangle)$, as we only define it for annotations without free annotation variables.

## 5.2 Internal & additional region arguments

Some functions may require additional region arguments, if the regions of the return type do not suffice. An example of such a function is `(.) :: (b -> c) -> (a -> b) -> a -> c`, which applies the first function after the second function. The function can be defined as follows:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = let y = g x in f y
```

In this example, we do not know where we should allocate the value stored in `y`. We could allocate `y` in the same region as the result (of type `c`), but that may cause that the lifetime of `y` is overestimated, for instance when passing `const 1` as `f`. Instead, we will add an additional region argument to both the annotation of `(.)` and the emitted function (after the region transform). This will become the first argument of `(.)`. At each usage of this function, we will pass an additional argument. This still allows the function to be used in higher order positions, and to be partially applied.

Note that additional region arguments are not related to polymorphism: the same problem occurs when substituting the type variables with for instance `Int`. However, polymorphism complicates the assignment of additional regions, as we cannot assign a polymorphic region sort to these regions. The additional region arguments are added before the quantifications. We cannot add the additional region arguments after the quantifications, as we do not know whether all usages of the function instantiate all quantifications.

The analysis works by first assigning additional region arguments to all subexpressions of the function. When normalizing the annotation, we try to remove some of those additional region arguments. Regions may either be removed as they do not escape out of the function (*internal regions*), or they may be unified with some other region.

## 5.3 Sorting rules

The sorting rules are the "typing rules" of the annotation language and describe the sorts of annotations. We write $\Gamma \vdash a : s$ to denote that annotation $a$ has sort $s$ in effect environment $\Gamma$ and we call an annotation *well sorted* if it has a sort.

### 5.3.1 Effect environment

To define the sorting rules, we must first consider the effect environment $\Gamma$, which contains information on the data types of the module and sorts of the annotation and region variables with their lifetime context which are in scope. A lambda adds an annotation variable to the effect environment, which we write as $\Gamma, \psi : s$. The addition of a region variable is written similarly, as $\Gamma, \rho : P^l$ or $\Gamma, \rho : P\langle\alpha\ \tau_1\ \ldots\ \tau_n\rangle^l$. The $l$ denotes the lifetime context of the lambda which introduced those region variables. We can only introduce region variables of a monomorphic sort or a polymorphic sort; variables of a tuple sort are not allowed. Instead, tuples should be bound to a tuple of region variables.

We write the introduction of a tree of region variables as $\Gamma +\!\!+ \hat{\rho} : \hat{P}^l$. For a monomorphic or polymorphic region sort, we add the region variable to the environment.

$$\begin{aligned} \Gamma +\!\!+ \rho : P^l &= \Gamma, \rho : P^l \\ \Gamma +\!\!+ \rho : P\langle\alpha\ \tau_1\ \ldots\ \tau_n\rangle^l &= \Gamma, \rho : P\langle\alpha\ \tau_1\ \ldots\ \tau_n\rangle^l \end{aligned} \tag{5.2}$$

For a tuple, we must add the region variables of each element of the tuple.

$$\begin{aligned} \Gamma +\!\!+ () : () &= \Gamma \\ \Gamma +\!\!+ (\hat{\rho}_1, \ldots \hat{\rho}_n) : \left(\hat{P}_1, \ldots \hat{P}_n\right)^l &= (\Gamma +\!\!+ \hat{\rho}_1 : \hat{P}_1^l) +\!\!+ (\hat{\rho}_2, \ldots \hat{\rho}_n) : \left(\hat{P}_2, \ldots \hat{P}_n\right)^l \end{aligned} \tag{5.3}$$

We write $\psi : s \in \Gamma$ to denote that variable $\psi$ has sort $s$ in environment $\Gamma$ and similarly $\rho : \hat{P}^l \in \Gamma$ for region variables.

### 5.3.2 Variables

The sort of an annotation variable is found in the effect environment.

$$\frac{\psi : s \in \Gamma}{\Gamma \vdash \psi : s} \tag{5.4}$$

For region variables, we also have subeffecting. It is namely allowed to use a variable with lifetime context local bottom as having any lifetime context, and we may use a monomorphic region variable as a polymorpic variable or as a tuple in applications.

$$\frac{\psi : s \in \Gamma}{\Gamma \vdash \psi : s} \tag{5.5}$$

$$\frac{\rho : \hat{P}^l \in \Gamma}{\Gamma \vdash \rho : \hat{P}^l} \tag{5.6}$$

$$\frac{\Gamma \vdash \rho : \hat{P}^{\uparrow\perp}}{\Gamma \vdash \rho : \hat{P}} \tag{5.7}$$

$$\frac{\Gamma \vdash \rho : P^l}{\Gamma \vdash \rho : \hat{P}^l} \tag{5.8}$$

Note that in the notation that we use, $P$ is syntax, namely the monomorphic region sort and $\hat{P}$ is variable, denoting a region sort. Rule 5.8 thus says that if a variable has the monomorphic region sort, then it may be used as any region sort.

As lambdas take a tree structure of region variables, we will also extend the region sorts to these structures.

$$\frac{\forall i.\ \Gamma \vdash \hat{\rho}_i : \hat{P}_i}{\Gamma \vdash (\hat{\rho}_1, \ldots \hat{\rho}_n) : \left(\hat{P}_1, \ldots \hat{P}_n\right)} \tag{5.9}$$

### 5.3.3 Lambdas & applications

For a lambda, we add the annotation argument and the region arguments with the lifetime context of the lambda to the type environment.

$$\frac{\Gamma, \psi : s_1 +\!\!+ \hat{\rho} : \hat{P}^l \vdash a : s_2}{\Gamma \vdash \lambda \left[\psi : s_1; \hat{\rho} : \hat{P}\right]_l \mapsto a : \left[s_1; \hat{P}\right] \to s_2} \tag{5.10}$$

We enforce that the argument of an application has the sort specified in the sort of the left hand side. The region arguments should have the appropriate lifetime context.

$$\frac{\Gamma \vdash a_1 : \left[s_1; \hat{P}\right]_l \to s_2 \quad \Gamma \vdash a_2 : s_1 \quad \Gamma \vdash \hat{\rho} : \hat{P}^l}{\Gamma \vdash a_1\ [a_2; \hat{\rho}]_l : s_2} \tag{5.11}$$

### 5.3.4 Lifetime relation

A lifetime relation consists of outlive constraints of the form $\rho \geq \rho'$. For each outlive constraint we require that if both operands are polymorphic, then they should be polymorphic on the same type. Furthermore, the left hand side must have lifetime context any and cannot be local bottom. We enforce this by first defining the sorting rules for lifetime relations with one outlive constraint.

$$\frac{\Gamma \vdash \rho : P \quad \Gamma \vdash \rho' : P^l}{\Gamma \vdash [\![\rho \geq \rho']\!] : R} \tag{5.12}$$

$$\frac{\Gamma \vdash \rho : P\langle \alpha\ \tau_1\ \ldots\ \tau_n\rangle \quad \Gamma \vdash \rho' : P^l}{\Gamma \vdash [\![\rho \geq \rho']\!] : R} \tag{5.13}$$

$$\frac{\Gamma \vdash \rho : P \quad \Gamma \vdash \rho' : P\langle \alpha\ \tau_1\ \ldots\ \tau_n\rangle^l}{\Gamma \vdash [\![\rho \geq \rho']\!] : R} \tag{5.14}$$

$$\frac{\Gamma \vdash \rho : P\langle \alpha\ \tau_1\ \ldots\ \tau_n\rangle \quad \Gamma \vdash \rho' : P\langle \alpha\ \tau_1\ \ldots\ \tau_n\rangle^l}{\Gamma \vdash [\![\rho \geq \rho']\!] : R} \tag{5.15}$$

The sorting rule for a lifetime relation with multiple constraints says that all constraints on their own should be valid.

$$\frac{n \geq 2 \quad \forall i.\ \Gamma \vdash [\![\rho_i \geq \rho_i']\!] : R}{\Gamma \vdash [\![\rho_1 \geq \rho_1', \ldots \rho_n \geq \rho_n']\!] : R} \tag{5.16}$$

### 5.3.5 Quantification and instantiation

To handle polymorphism of the source language, the annotation language and sorts have quantifications. The argument of an instantiation must have a quantified sort, which we will then instantiate using the given type.

$$\frac{\Gamma \vdash a : s}{\Gamma \vdash \forall \alpha.\ a : \forall \alpha.\ s} \tag{5.17}$$

$$\frac{\Gamma \vdash a : \forall \alpha.\ s}{\Gamma \vdash a\ \{\ \tau\ \} : s[\alpha := \tau]_\Gamma} \tag{5.18}$$

### 5.3.6 Tuples

Tuple annotations get the tuple sort, with the sorts of all elements. The projection takes an element from a tuple. Note that the projection is zero-based.

$$\frac{\forall i.\ \Gamma \vdash a_i : s_i}{\Gamma \vdash (a_1, \ldots a_n) : (s_1, \ldots s_n)} \tag{5.19}$$

$$\frac{\Gamma \vdash a : (s_0, \ldots s_{n-1}) \quad 0 \leq k < n}{\Gamma \vdash \pi_k(a) : s_k} \tag{5.20}$$

### 5.3.7 Join and bottom

Both operands of a join should have the same sort, which is also the sort of the join itself.

$$\frac{\Gamma \vdash a_1 : s \quad \Gamma \vdash a_2 : s}{\Gamma \vdash a_1 \sqcup a_2 : s} \tag{5.21}$$

The bottom annotation may have any sort.

$$\frac{}{\Gamma \vdash \bot : s} \tag{5.22}$$

### 5.3.8 Fixpoint combinators

The fixpoint combinator takes a function of sort $[s, ()] \to s$ and evaluates to its least fixpoint, which has sort $s$.

$$\frac{\Gamma \vdash a : [s, ()] \to s}{\Gamma \vdash fix : s.\ a\ :\ s} \tag{5.23}$$

The fixpoint combinator with escape check has more constraints. Sort $s$ should be a function taking additional region arguments ($\hat{P}$, which were introduced in chapter 5.2), and returning an annotation of the sort of a function whose arity is at least $n$.

$$\frac{\Gamma \vdash a : [s; ()] \to s \quad s = \left[(); \hat{P}\right] \to \Psi_\Gamma(\tau) \quad arity(\tau) \geq n}{\Gamma \vdash fix\ escape\ \left[n; \hat{P}\right] : s.\ a\ :\ s} \tag{5.24}$$

## 5.4 Evaluation rules

We will introduce the evaluation rules of the annotation language in this section. We write $a \longrightarrow a'$ to denote that a well sorted annotation $a$ evaluates to $a'$. In these evaluation rules, we will not consider name collisions and alpha conversion. In our implementation we namely used De Bruijn indices [15], which prevents name collisions and make normalisation easier as there is no freedom in choosing names. For clarity, we however do not use De Bruijn indices as examples are more clear with proper variable names.

The "evaluates to" relation is reflexive and transitive.

$$\frac{}{a \longrightarrow a} \tag{5.25}$$

$$\frac{a_1 \longrightarrow a_2 \quad a_2 \longrightarrow a_3}{a_1 \longrightarrow a_3} \tag{5.26}$$

### 5.4.1 Sub-annotations

The subexpressions (sub-annotations) of annotations may be evaluated.

$$\frac{a \longrightarrow a'}{\lambda \left[\psi : s; \hat{\rho} : \hat{P}\right]_l \mapsto a \longrightarrow \lambda \left[\psi : s; \hat{\rho} : \hat{P}\right]_l \mapsto a'} \tag{5.27}$$

$$\frac{a_1 \longrightarrow a_1' \quad a_2 \longrightarrow a_2'}{a_1\ \ [a_2; \hat{\rho}]_l \longrightarrow a_1'\ \ [a_2'; \hat{\rho}]_l} \tag{5.28}$$

$$\frac{a \longrightarrow a'}{\forall \alpha.\ a \longrightarrow \forall \alpha.\ a'} \tag{5.29}$$

$$\frac{a \longrightarrow a'}{a\ \{\tau\} \longrightarrow a'\ \{\tau\}} \tag{5.30}$$

$$\frac{\forall i.\ a_i \longrightarrow a_i'}{(a_1, \ldots a_n) \longrightarrow (a_1', \ldots a_n')} \tag{5.31}$$

$$\frac{a \longrightarrow a'}{\pi_i(a) \longrightarrow \pi_i(a')} \tag{5.32}$$

$$\frac{a_1 \longrightarrow a_1' \quad a_2 \longrightarrow a_2'}{a_1 \sqcup a_2 \longrightarrow a_1' \sqcup a_2'} \tag{5.33}$$

### 5.4.2 Tuples

A projection extracts an element from a tuple. The index is zero-based.

$$\overline{\pi_i((a_1, \ldots a_n)) \longrightarrow a_{i+1}} \tag{5.34}$$

### 5.4.3 Join

A join of two lifetime relations evaluates to the smallest lifetime relation containing both operands. The algorithm that we use is discussed in chapter 9.2

$$\frac{[\![\rho_1^\circ \geq \rho_1^\bullet, \ldots]\!] \text{ is the smallest lifetime relation satisfying } \rho_i \geq \rho_i' \text{ for } i = 1\ \ldots\ n+m}{[\![\rho_1 \geq \rho_1', \ldots \rho_n \geq \rho_n']\!] \ \sqcup\ [\![\rho_{n+1} \geq \rho_{n+1}', \ldots \rho_{n+m} \geq \rho_{n+m}']\!] \longrightarrow [\![\rho_1^\circ \geq \rho_1^\bullet, \ldots \rho_k^\circ \geq \rho_k^\bullet]\!]} \tag{5.35}$$

The join is associative, commutative and idempotent.

$$\overline{(a_1 \sqcup a_2) \sqcup a_3 \longrightarrow a_1 \sqcup (a_2 \sqcup a_3)} \tag{5.36}$$

$$\overline{a_1 \sqcup a_2 \longrightarrow a_2 \sqcup a_1} \tag{5.37}$$

$$\overline{a \sqcup a \longrightarrow a} \tag{5.38}$$

Bottom is the identity of the join.

$$\overline{\bot \sqcup a \longrightarrow a} \tag{5.39}$$

Furthermore, we have some rules on how joins and bottoms distribute over other annotations. For quantifications and lambdas, we move the union inward. With alpha conversion, we can assure that the type variable, region variables and annotation variable have the same name on both sides. As we used De Bruijn indices in the implementation, naming is no problem and we do not need to perform renaming here.

$$\overline{(\forall \alpha.\ a_1) \sqcup (\forall \alpha.\ a_2) \longrightarrow \forall \alpha.\ a_1 \sqcup a_2} \tag{5.40}$$

$$\overline{\left(\lambda \left[\psi : s; \hat{\rho} : \hat{P}\right]_l \mapsto a_1\right) \sqcup \left(\lambda \left[\psi : s; \hat{\rho} : \hat{P}\right]_l \mapsto a_2\right) \longrightarrow \lambda \left[\psi : s; \hat{\rho} : \hat{P}\right]_l \mapsto a_1 \sqcup a_2} \tag{5.41}$$

$$\overline{\forall \alpha.\ \bot \longrightarrow \bot} \tag{5.42}$$

$$\overline{\lambda \left[\psi : s; \hat{\rho} : \hat{P}\right]_l \mapsto \bot \longrightarrow \bot} \tag{5.43}$$

For applications and instantiations, we move the joins outward.

$$\overline{(a_1 \sqcup a_2)\ [a, \hat{\rho}]_l \longrightarrow a_1\ [a, \hat{\rho}]_l\ \sqcup\ a_2\ [a, \hat{\rho}]_l} \tag{5.44}$$

$$\overline{(a_1 \sqcup a_2)\ \{\tau\} \longrightarrow a_1\ \{\tau\}\ \sqcup\ a_2\ \{\tau\}} \tag{5.45}$$

$$\overline{\bot\ [a, \hat{\rho}]_l \longrightarrow \bot} \tag{5.46}$$

$$\overline{\bot\ \{\tau\} \longrightarrow \bot} \tag{5.47}$$

### 5.4.4 Application

An application can be evaluated if it targets an abstraction. We then apply a substitution to the body of the abstraction.

$$\frac{a_1[\psi := a_2][\hat{\rho}_1 := \hat{\rho}_2]}{\left(\lambda \left[\psi : s; \hat{\rho}_1 : \hat{P}\right]_l \mapsto a_1\right)\ [a_2; \hat{\rho}_2]_l} \tag{5.48}$$

The substitution of the annotation argument $\psi$ is a syntactic substitution (ignoring name collisions). The substitution of region variables is slightly different, as a lambda takes a tree of region variables. As noted before, a single region $\rho$ may also be passed to a tree of region variables, which means that all variables in that tree are substituted with $\rho$. Formally, we can define this substitution in terms of the ordinary syntactic substitution $(a[\rho := \rho'])$.

$$\begin{aligned} a[(\hat{\rho}_1, \ldots \hat{\rho}_n) := (\hat{\rho}'_1, \ldots \hat{\rho}'_n)] &= a[\hat{\rho}_1 := \hat{\rho}'_1] \ldots [\hat{\rho}_n := \hat{\rho}'_n] \\ a[(\hat{\rho}_1, \ldots \hat{\rho}_n) := \rho] &= a[\hat{\rho}_1 := \rho] \ldots [\hat{\rho}_n := \rho] \\ a[\rho := \rho'] &= \text{the syntactic substitution of } \rho \text{ with } \rho' \text{ in } a \end{aligned} \tag{5.49}$$

### 5.4.5 Instantiation

An instantiation of a quantification, $(\forall \alpha.\ a)\ \{\tau\}$, can be evaluated, by essentially substituting type variable $\alpha$ with type $\tau$ in $a$. We already defined such instantiation of types, region sorts and annotation sorts. As seen in the instantiation rules for region sorts, it may occur that a lambda gets more or fewer region arguments, when a polymorphic region sort is replaced by the regions of the instantiated type. This would require some additional notation to define this formally, so we instead introduce it informally in text.

The substitution is essentially propagated through the annotation. Only lambdas are special. Consider some region variable $\rho$ of a polymorphic region sort $P\langle\alpha\ \tau_1\ \ldots\ \tau_n\rangle$, which is polymorphic in the type variable which is substituted. We find the sort of the region variable after instantiation with $\hat{P} = P^{\circ}_{\Gamma}(\alpha\ \tau_1\ \ldots\ \tau_n)$. We introduce new region variables $\hat{\rho}$, based on the tree $\hat{P}$. We now must substitute all occurrences of $\rho$ in the body of the lambda. For usages of $\rho$ in an application, we replace $\rho$ with $\hat{\rho}$, the tree of new region variables, as an application may contain such tree structure.

For occurrences of $\rho$ in an outlive constraint, of the form $\rho \geq \rho'$ or $\rho' \geq \rho$, we will extend the outlive constraint elementwise. We distinguish two cases. If both operands of the outlive constraints are polymorphic, then they must be polymorphic on the same type, which is now being substituted. We create new outlive constraints, element wise, matching region variables from the same positions of the two trees of region variables. Consider that $\rho$ is substituted with $(\rho_1, \rho_2)$ and $\rho'$ with $(\rho'_1, \rho'_2)$. The constraint $\rho \geq \rho'$ is then substituted with $\rho_1 \geq \rho'_1, \rho_2 \geq \rho'_2$.

If only one of the operands of the outlive constraint is monomorphic, we also create new outlive constraints element wise. For instance, consider a constraint $\rho \geq \rho'$ where $\rho$ is substituted with $(\rho_1, \rho_2)$. We then create the constraints $\rho_1 \geq \rho'$ and $\rho_2 \geq \rho'$.

### 5.4.6 Fixpoints

First we consider ordinary fixpoints, without escape check. A fixpoint contains a function, $a$, which has sort $[s; ()] \to s$. Fixpoints are evaluated by repeatedly applying that function, until a fixpoint is reached. We do so by repeatedly taking the function composition $a \circ a$. First we introduce a rule to perform one iteration.

$$\frac{}{fix : s.\ a \longrightarrow fix : s.\ \lambda[\psi : s; () : ()] \mapsto a\ [a\ [\psi; ()]; ()]} \tag{5.50}$$

$$\frac{a \longrightarrow a'}{fix : s.\ a \longrightarrow fix : s.\ a'} \tag{5.51}$$

When a fixpoint is reached, we can break out of the fixpoint.

$$\frac{a\ [\bot; ()] \longrightarrow a' \quad a\ [a\ [\bot; ()]; ()] \longrightarrow a'}{fix : s.\ a \longrightarrow a'} \tag{5.52}$$

The rules for the fixpoint with escape check are similar, but we will try to remove additional region arguments there. This procedure is described in chapter 5.6. A fixpoint with escape check can only be evaluated if the body has no free variables, as free variables would cause the escape check to be inconsistent.

### 5.4.7 Normal form

If an annotation cannot be evaluated further, we say that it is in normal form. As the join is commutative and associative, we enforce an arbitrary order on annotations and order the annotations in a join accordingly. Furthermore, variables and fixpoints must be saturated, which we ensure by eta expansion.

## 5.5 First order fixpoints

With any fixpoint iteration, one may ask whether a fixpoint will eventually be reached. For region inference, this is not the case. Higher order functions which take a function as argument and apply that function repeatedly on its result are problematic. For instance, the analysis does not find a fixpoint for the functions `fix`, `foldr`, `foldl`, `iterate` and `until` from the Prelude of Haskell.

A similar problem occurred in previous work on higher order analyses by Thorand [51]. In his context, the problem was that there was no canonical form known and the comparisons of annotation terms was thus not possible. The annotation would reach a fixpoint, but this could not be concluded by a syntactic comparison. As a solution, all possible inputs to the functions were tried and their results were checked for equivalence. The set of all inputs was finite, as their source language did not have polymorphism. As our language does have polymorphism, this set becomes infinite. Furthermore, even for monomorphic functions this approach would be very expensive as our analysis considers a much larger domain than the dependency analysis by Thorand [51].

Besides the comparison, we also have the problem that polymorphic functions may not even have a fixpoints. Polymorphic functions exist such that given any number, there is an instantiation of the type such that the fixpoint iteration requires more iteration than the given number. This can be shown by passing a function which iterates a tuple like $(a_1, \ldots a_n) \mapsto (a_n, a_1, a_2, \ldots a_{n-1})$ to a higher order function like *iterate*. After $n$ iterations, each element has been on all positions. As we can make $n$ arbitrarily large, there is no fixed number of iterations which will result in a fixpoint.

As a partial solution to this problem, we introduce the notion of a *first order fixpoint*. A fixpoint $fix : s.\ a$ is in a first order fixpoint if $a\ [\bot; ()] \longrightarrow a_1$, $a\ [a\ [\bot; ()]; ()] \longrightarrow a_2$,

$a_1$ and $a_2$ cannot be evaluated further, and evaluate to the same term after substituting all variables with bottom. When a first order fixpoint is reached, we do not evaluate the fixpoint further. This is a restriction for the code generation, as the annotation contains a fixpoint. The first order parts of the function are however fixed, so we can reason with those. We cannot reason about the higher order parts, the function calls, but that is not that much of a restriction as we can already do little reasoning on higher order function calls.

We use first order fixpoints similarly on a fixpoint with escape check. However, when a first order fixpoint is reached, we convert the fixpoint with escape check to an ordinary fixpoint; the escape check is based on the first order fixpoint.

The fixpoints are preserved, until we have more information on the use of the function. For instance, we may be able to evaluate a fixpoint further if we know the annotation of an argument passed to a higher order function. However, we have not found a general and consistent way to make use of the additional information. What we currently do is to check which arguments are kept constant, in the recursive calls of the fixpoint. We inline those arguments, which may give more information on the fixpoint. This way we can for instance analyse functions defined in terms of `fix`, or actually most higher order functions as they usually tend to keep such arguments constant.

Besides of this inlining, we will also try to split a fixpoint if it contains tuples, as we may be able to partially evaluate such fixpoint. We do so by first lifting the lambdas to the top, even out of lambda abstractions. This can be done by adding projections explicitly. We then create fixpoints per element of the tuples. However, as the elements of the tuple may reference other elements, we must inline the fixpoints multiple times, which may cause a factorial number of fixpoints. In practise this transform will be beneficial for the performance, as many of those fixpoints can directly be evaluated.

These transformations are not sufficient to always evaluate a fixpoint. It would be desired that we will always find an annotation without fixpoint if we analyse a first order function even if the function calls higher order functions. Evaluation strategies to better handle fixpoints are left as future work. A possible alternative would be to widen the annotation and get an annotation which is an upperbound of the fixpoint. This will need to be done in a consistent way, such that mutual recursive functions are analysed properly.

## 5.6 Removing additional region arguments

In the fixpoint combinator with escape check, we try to remove additional region arguments. We do this with an escape check, by finding cycling outlive constraints and by collapsing.

### 5.6.1 Escape check

The escape check finds regions which do not escape out of a function and can thus be allocated within that function. A set of additional region arguments $D$ may be allocated within the function if there are no constraints of the form $\rho_1 \geq \rho_2$ with $\rho_1 \in D$ and $\rho_2 \notin D$. The escape check finds the largest set $D$. The implementation of the escape check, together with other algorithms on lifetime relations, is discussed in chapter 9.

### 5.6.2 Cyclic outlive constraints

When an annotation gives the outlive constraints $\rho_1 \geq \rho_2$ and $\rho_2 \geq \rho_1$ and $\rho_1$ is an additional region argument, we remove $\rho_1$ by substituting it with $\rho_2$. This decreases the number of region arguments, which also reduces the runtime overhead of these region arguments, and also has the benefit that the region arguments are passed later for partially

applied functions. Additional region arguments are passed with the direct use of the function, thus before passing the first argument. This means that the additional region arguments are instantiated the same for all calls to a partially applied function. Normal region arguments for the last argument of a function may have different instantiations for each call. This limitation of the additional region arguments is caused by the code generation, as we need to use the results of the analysis for code generation and need to handle partial applications in a uniform way. Substituting additional region arguments with other regions will thus reduce the sharing between different calls of a partially applied function. Furthermore, it also reduces the sizes of lifetime relations, which may improve the time and memory usage of the analysis.

### 5.6.3 Collapsing

Collapsing is another method to reduce the number of additional region arguments. Given two regions $\rho_u$ and $\rho_v$ such that $\rho_u \geq \rho_v$, we will substitute $\rho_u$ with $\rho_v$ if $\rho_u$ is an additional region argument and the substitution will not add additional constraints on $\rho_v$. As the latter condition is rather imprecise, we will formalize that now by introducing the term "directly outlives".

**Definition 5.6.1** (Outlive set)**.** *The outlive set* $outlive(\rho_u)$ *of a region* $\rho_u$ *in a lifetime relation* $R$ *is the set of all regions* $\rho_v \neq \rho_u$ *such that* $\rho_u \geq \rho_v$*.*

Note that the outlive set is the predecessor set of the relation treated as a graph, where loops (reflexivity) are ignored.

**Definition 5.6.2** (Directly outlives)**.** *We say that region* $\rho_u$ *directly outlives region* $\rho_v$ *if* $outlive(\rho_u) = outlive(\rho_v) \cup \{v\}$*.*

Note that a region directly outlives at most one region. In first order functions, we can find regions which directly outlive some other region by looking at the relation specified in the normalized annotation. For higher order functions, we do not yet know all constraints between regions. We take the main relation from the annotation, which corresponds to a saturated call to the function, and will substitute region $\rho_u$ with $\rho_v$ if $\rho_u$ directly outlives $\rho_v$ in that relation, $\rho_u$ is an additional region argument and $\rho_u$ does not occur elsewhere in the annotation except for applications marked with lifetime context $\uparrow\!\!\perp$. The region may be used in applications with $\uparrow\!\!\perp$, as that means that the region variable may only be used on the right hand side of the outlives constraint ($\geq$), thus the outlives set of $\rho_u$ will not change.

Furthermore, we note that annotations also contain relations for partially applying a function. These annotations say that the additional region arguments and the regions corresponding to function arguments must outlive the thunk. This would prevent that additional region arguments are removed; thus these relations must be ignored for the collapsing.

## 5.7 Examples

### 5.7.1 `id` specialized for Ints

As a simple example, we start with an identity function specialized to integers.

```
idInt :: Int -> Int
idInt x = x
```

The analysis produces the following annotation on the identity function:

$$
\begin{array}{l}
\lambda[();()] \mapsto \\
\quad \lambda[();(\rho_{thunk} : P, \rho_{value} : P, ())] \mapsto \\
\quad\quad ( \\
\quad\quad\quad \lambda[();\rho_{previous\_thunk} : P] \to \lambda[();(\rho_{result} : P, ())]_{\uparrow} \to \\
\quad\quad\quad\quad [\![\ \rho_{value} \geq \rho_{result} ]\!], \\
\quad\quad\quad () \\
\quad\quad )
\end{array}
$$

The resulting annotation starts with a lambda taking no arguments, as this identity function does not require additional region arguments. On the second line, we see another lambda. This lambda consumes the annotation arguments and the region arguments of argument `x` of `idInt`. An integer does not contain any functions, so its annotation sort is $()$. An integer has two region arguments, namely one for the thunk and one for the value if it is evaluated[1].

The third line contains two lambdas, which take the regions of the previous thunk and the result. The previous thunk is only relevant when partially applying a function, which cannot be done with a function of arity 1. The region of the thunk is used in the resulting lifetime relation, which says that the region of the argument should outlive the region of the returned value. This constraint originates from subeffecting. We pretend that the returned value is stored in region $\rho_{result}$, whereas it is actually stored in region $\rho_{value}$. We are only allowed to do so if we also give the constraint $\rho_{value} \geq \rho_{result}$.

Note that the annotation puts no constraints on the thunk region of the argument, $\rho_{thunk}$. The identity function is strict in the first argument, which means that the argument will be evaluated before returning a value. The identity function thus does not put a constraint on the thunk region.

### 5.7.2 `id`

To illustrate annotations with polymorphism, we will now generalize the identity function to arbitrary types.

```
id :: a -> a
id x = x
```

$$
\begin{array}{l}
\lambda[();()] \mapsto \forall\alpha.\ \lambda[\psi_1 : \Psi\langle\alpha\rangle;(\rho_{thunk} : P, \rho_{value} : P, \rho_{nested} : P\langle\alpha_0\rangle)] \mapsto ( \\
\quad \lambda[();\rho_{previous\_thunk} : P] \to \lambda[();(\rho_{result} : P, \rho_{result\_nested} : P\langle\alpha\rangle)]_{\uparrow} \to \\
\quad\quad [\![\ \rho_{value} \geq \rho_{result}, \rho_{nested} \geq \rho_{result}, \\
\quad\quad\quad \rho_{nested} \geq \rho_{result\_nested}, \rho_{nested} \geq \rho_{value}\ ]\!], \\
\quad \psi_1)
\end{array}
$$

Similar to the monomorphic identity function, `id` does not have additional region arguments. After the lambda consuming the additional region arguments, we see a quantification, as `id` is polymorphic. The next lambda now takes both the annotation of the argument, of sort $P\langle\alpha_0\rangle$, and the region arguments. We now have more region arguments than with the monomorphic identity function, as we have a polymorphic region argument for the nested fields of `x`. This causes that we also have more constraints:

[1] Integers are in practise passed by value instead of by reference, as they fit in a word. For this example, we don't consider that and treat them as allocated objects.

- We have one additional subeffecting constraint: $\rho_{nested} \geq \rho_{result\_nested}$.
- We get a containment constraint, saying that the nested fields must outlive the object: $\rho_{nested} \geq \rho_{value}$.
- Transitivity gives yet another constraint: $\rho_{nested} \geq \rho_{result}$.

Besides the lifetime relation, the annotation also returns annotation $\psi_1$, which is the annotation of the returned value. This annotation may be used when the instantiated type of $\alpha$ is or contains a function.

### 5.7.3 `$`

We will now consider a higher order function, and as annotations will quickly increase in size, we choose `$`. This operator takes a function `f` and an argument `x` and applies `f` to `x`.

```
($) :: (a -> b) -> a -> b
($) f x = f x
```

$$
\begin{aligned}
&\lambda[();()] \mapsto \forall\alpha_1.\forall\alpha_2.\\
&\quad \lambda[\psi_f : s; (\rho_{f\_thunk} : P, \rho_f : P, ())] \mapsto (\\
&\qquad \lambda[(), \rho_{previous\_thunk} : P] \mapsto \lambda[(), (\rho_{result} : P, ())]_{\uparrow\perp} \mapsto\\
&\qquad\quad [\![\rho_{previous\_thunk} \geq \rho_{result}, \rho_{f\_thunk} \geq \rho_{result}, \rho_f \geq \rho_{result}]\!],\\
&\qquad \lambda[\psi_x : \Psi\langle\alpha_1\rangle; (\rho_{x\_thunk} : P, \rho_x : P, \rho_{x\_nested} : P\langle\alpha_1\rangle)] \mapsto (\\
&\qquad\quad \lambda[(), \rho_{previous\_thunk} : P] \mapsto \lambda[(), (\rho_{result} : P, \rho_{result\_nested})]_{\uparrow\perp} \mapsto\\
&\qquad\qquad [\![\rho_{x\_nested} \geq \rho_x]\!] \sqcup\\
&\qquad\qquad \pi_0(\psi_f\ [\psi_x; (\rho_{x\_thunk}, \rho_x, \rho_{x\_nested})])\ [();\rho_f]\ [(); (\rho_{result}, \rho_{result\_nested})]_{\uparrow\perp},\\
&\qquad\quad \pi_1(\psi_f\ [\psi_x; (\rho_{x\_thunk}, \rho_x, \rho_{x\_nested})])\\
&\qquad )\\
&\quad )
\end{aligned}
$$

where $s = [\Psi\langle\alpha_1\rangle; (P, P, P\langle\alpha_1\rangle)] \to [();P] \to [(); (P, P\langle\alpha_2\rangle)]_{\uparrow\perp}$

The annotation starts with consuming zero additional region arguments and two quantifications. It then consumes the annotation of argument `f`, which has sort $s$. After consuming the annotations and regions of `x`, the annotation returns the constraints which the saturated call to `$` enforces and the annotations on the returned value. Both parts contain a call to $\psi_f$. We pass the region and annotation arguments of `x` to $\psi_f$, which will then give the constraints that the call itself gives and the annotations of the returned value.

### 5.7.4 Partial application

As noted before, we pass a region pointing at the previous thunk to handle partial applications. As partial applications become interesting if the function takes at least three arguments and annotations will quickly increase in size, we will not give the annotation of such a function but instead walk through effects of the annotation. We consider function `f` with arity three, to which we pass the arguments one by one.

```
f x y z = ...

a = f 1
b = a 2
c = b 3
```

We denote the regions of `a`, `b` and `c` by respectively $\rho_a$, $\rho_b$ and $\rho_c$. First we look at the effects of the previous thunk region. The previous thunk argument for the first application (stored in `a`) is the region of `f`. As that is a global function, which will thus not be deallocated during the lifetime of the program, that region is $\rho_{global}$. We will thus give the constraint $\rho_{global} \geq \rho_a$, which we already implicitly have.

For the application passing the second argument, the previous thunk region is $\rho_a$. We thus yield the constraint $\rho_a \geq \rho_b$. Finally, when passing the third argument, we do not use the previous thunk as the call is saturated. Region $\rho_3$ thus does not put any constraints on the lifetimes of $\rho_a$ and $\rho_b$.

The annotations for the partial applications will also ensure that the regions of the arguments outlive the region of the thunk. Thus, the regions of expression `1` must outlive $\rho_a$ and the regions of `2` must outlive $\rho_b$. Because of transitivity, we have that the regions of `1` also outlive $\rho_b$, so the second partial applied value will also retain the regions of the second argument.

If function `f` has additional region arguments, they will be passed before passing the regions and annotations of the first argument. They will thus be stored in thunk $a$ at runtime and we thus need to give constraints that each additional region argument outlives $\rho_a$. Because of transitivity, we will then also get constraints that the additional region arguments outlive $\rho_b$, thus the second partial applied value will also keep the additional region arguments in memory.

# 6

# Backend of the Helium compiler

The region inference algorithm was implemented in the new, LLVM [35] based backend [16] of the Helium compiler. Helium [22] is a compiler for Haskell, mainly focussed on generating high quality type errors [23]. In this section, we will introduce the pipeline of the backend of the compiler and the intermediate language Iridium. After introducing the backend, we will give details on how we construct region annotations for the Iridium language in chapter 7. We discuss the interaction of region inference with other compiler optimizations which are performed in the Helium compiler in chapter 8.

The compiler parses Haskell source files to a Haskell AST, on which type checking is performed. After type checking, the program is compiled to Core, which is a simple typed functional language with pattern matching. Core is typed using the type system introduced in chapter 1.3. After several passes, Core is compiled to Iridium, which features the same type system, but has imperative control flow. Functions must be top level, may have multiple arguments and can be partially applied. It supports data types with pattern matching [16].

## 6.1 Iridium

Iridium is the last intermediate language of the new backend, before we compile to LLVM. An Iridium file consists of data types, methods, type synonyms and signatures of imported methods. Type classes are represented as data types in this phase. A data type is defined using the type signatures of its constructors. Signatures of imported methods are called abstract methods and are annotated with the module name in which they are defined and the previously computed region annotation.

A method may have type and variable arguments. The arity of a function is the number of value arguments that the function expects. A method can be called by giving all arguments. For laziness and partial applications, we can also create a thunk, which represents a call to a method. If a thunk contains all arguments to a function, we say that the thunk is *saturated* and it can be further evaluated to weak head normal form (WHNF). If not enough arguments were passed, the thunk is *undersaturated* and represents a partial application. Remaining arguments can be passed in later applications. A thunk may also contain more arguments than the method expects, which we call an *oversaturated* thunk. When evaluating such a thunk, we first call the functions with the first part of the arguments, such that the call is saturated, which gives a thunk representing a partial application. That resulting thunk is then applied to the remaining arguments.

Expressions in Haskell which are not a function, are compiled to functions with zero arguments. We globally create a thunk for such method, which is shared between all of the usages of the method. We distinguish global and local variables. Global variables, denoted by `@name` : $\tau$, are used to refer to methods and local variables, `%name` : $\tau$, are local to a method.

Strict values are all represented in one word. We currently only support 64 bit architectures, thus words are always 64 bits, but the compiler may be extended to more architectures. Constructors with fields are heap allocated and referenced by pointer. Constructors without fields are not allocated, but are directly stored in a word. We put a 1 in the last bit to distinguish them from pointers. Integers are also stored in a word as they fit in 64 bits. Non-strict values require an additional bit, which denotes whether the value is in weak head normal form (WHNF). If that bit is false, the word points at a thunk object. If it is true, the word contains the value, represented the same way as strict values.

A method consists of blocks. The control flow in a block is linear, except for function calls, similar to the control flow in LLVM [1]. The control flow between blocks is handled by (conditional) jumps. The execution of a method starts in the first block. Other blocks may be reached by those (conditional) jumps. A jump cannot target the first block.

## 6.2 Static Single-Assignment form

Variables in Iridium can only have a single assignment[1], similar to LLVM [1]. This is called Static Single-assignment form, or SSA for short [5]. Without the guarantee of having only a single assignment, it is harder to reason about programs, as there may be multiple assignments which lead to the value of some variable. In cases where multiple possible assignments are desired, this is made explicit using a phi node.

A phi node describes a join of dataflow. It assigns a value to a variable, depending on the previous block of the execution. In Iridium, we write it as `%x = phi (b1 => y1: t, b2 => y2: t, ...)` and it has the semantics that variable `x` gets the value of $\texttt{y}_i$, if $\texttt{b}_i$ was the previous branch of the execution.

As the source language Haskell does not support updates of variables, we do not need those phi nodes for variable updates. We do need them to compile pattern matching and we also use them for tail call optimization (chapter 8.3). For pattern matching, the code branches on the patterns of the alternatives. After branching, the control flow will join. Consider `x = case a on True -> f; False -> g`. We need to assign a value to `x`, depending on the branch we took. This is done using a phi node and it is written in Iridium as follows:

```
entry:
  case %a: !Bool constructor (
    @True[0]: Bool to branchtrue,
    @False[0]: Bool to branchfalse)
branchtrue:
  %y1 = eval %f: Int
  jump end
branchfalse:
  %y2 = eval %g: Int
  jump end
end:
```

[1]This chapter is adapted from the report of an experimentation project we previously did on the Helium compiler.

```
%x = phi (
  branchtrue => %y1: !Int,
  branchfalse => %y2: !Int)
```

SSA is an alternative to the lambda calculus, as a way to represent control flow or data flow. Appel [5] noted that they are very similar. The advantages of SSA for the Helium compiler are that LLVM is also using SSA and that it can express more (namely, loops). With respect to the region inference this means that we can perform the analysis on a level closer to the LLVM code.

## 6.3 Instructions

Iridium has various instructions; first we consider the instructions which are used as the end of a block.

**return** A return instruction, `return` *var*, returns a value from a method and gives control back to the previous method on the call stack.

**jump** A jump instruction, `jump` *block*, unconditionally jumps to a block.

**case** A case instruction, `case` *var alts*, conditionally jumps to a block based on the constructor of the argument, in case of a data type, or the value, in case of an integer.

**unreachable** The unreachable, written as `unreachable` or `unreachable` *var*, instruction denotes that some location in the code in unreachable. It may optionally contain a variable, if the computation of that variable causes that this location is unreachable. This prevents that dead code elimination would for instance remove a call to the `error` function of the Prelude.

Furthermore we have instructions, which may be used before these terminal instructions.

**let** A let instruction is written as $\%name = expr$. It (strictly) evaluates an expression and binds its result to a variable. Expressions are presented in chapter 6.4.

**let alloc** A let alloc instruction, denoted by `letalloc` $\overline{bind}$, allocates thunk or constructor objects. The thunks or objects may be (mutual) recursive. A bind is either a function call to construct a thunk, targeting a method or a different thunk, or a constructing invocation.

**match** A match instruction extracts fields from a constructor or tuple. The behaviour is undefined if the object is of a different type as what the match instruction says. It is explicit in the instantiation of the type variables of the constructor or the types of the elements of a tuple.

## 6.4 Expressions

The let expression contains an expression, which is evaluated and bound to a variable. Iridium has the following expressions:

**literal** Denotes an integer, float or string literal.

**call** A call expression, `call` $@name[n]\left(\overline{\{\tau\} \mid var}\right)$, calls a method. The method name is annotated with its arity and type. The call can provide both type and value arguments.

**instantiate** An instantiate expression, `instantiate` $var\ \tau$ instantiates a polymorphic value.

**eval** An evaluate expression, `eval` $var$, takes a non-strict value and evaluates it to WHNF. If the value was already evaluated, it yields the previously computed value.

**castthunk** The castthunk expression, `castthunk` $var$, takes a strict value and converts it to a non-strict value.

**var** A variable expression, `var` $var$, evaluates to the value of the argument. It does not evaluate the value, thus if the argument is non-strict, the result will also be non-strict.

**phi** A phi expression, `phi` $\left(\ \overline{block \Rightarrow var}\ \right)$, represents a phi node in the control flow graph.

**seq** The sequence expressions, `seq` $var_1 var_2$ is used to mark a dependency of $var_1$. It yields the value of $var_2$. It implies that $var_1$ is life if the resulting value of this expression is used, which prevents that $var_1$ is removed by dead code elimination.

**undefined** The undefined expression, `undefined` $\tau$, yields an arbitrary value. It has no further semantics. When using such value, the program may crash. Note that this is an undefined value, similar to the `undefined` instruction of LLVM [1], not the Haskell function `undefined`. It is for instance in method calls to methods which do not use an argument.

**primitive** Primitive expressions are used for primitive operations like integer addition, comparisons and so on.

## 6.5 Example

As an example, we consider `head` from the Prelude. This function takes a list and returns the first element of the list. If the list was empty, it throws an error.

```
head :: [a] -> a
head (x:_) = x
head _ = error "Prelude.head: empty list"
```

This function is compiled to the following Iridium method, slightly modified for readability.

```
export_as @head define @Prelude#head: { (forall a. !([a]) -> a) }
  $ (forall a, %xs: !([a])): a [trampoline] {

entry:
  case %xs: !([a]) constructor (
    @":": (forall a. a -> [a] -> [a]) to match_case_cons,
    @"[]": (forall a. [a]) to match_case_nil)

match_case_cons:
  match %xs: !([a]) on @":": (forall b. b -> [b] -> [b]) {a} (%x, _)
  %x_value = eval %x: a
  return %x_value: !a

match_case_nil:
  %msg = literal str "Prelude.head: empty list"
  %msg_thunk = castthunk %msg: !String

  %result = call @LvmException#error[1]: (forall v$0. [Char] -> v$0)
    $ ({a}, %msg_thunk: [Char])

  unreachable %result: !a
}
```

The header of the method contains the name of the function. The method has both a name which is only used for the backend and should be unique, `Prelude#head`, and an exported name, `head`, used for the module system. Functions which are not exported do not get an exported name. The header also has a type signature and the list of arguments.

The function is strict in its argument. We pattern match on it into two branches. The first branch is taken when the argument is a cons object. We then extract the first element from the list, store it in `%x` and evaluate it.

The second branch is taken when the argument is an empty list. We then call the `error` function, which will crash the program. The terminal instruction is thus not reachable, hence we use `unreachable`.

# 7

# Region inference on Iridium

In this section, we will consider the generation of (non-normalized) annotations. As we have a rich annotation language, we can quickly generate verbose annotations and let the evaluation rules simplify the annotations. The analysis is performed per binding group, which contains a set of mutual recursive functions, similar to type inferencing in Helium [23]. Binding group are ordered such that there are no forward references, where a method in a binding group uses a method defined in a later binding group.

The analysis per binding group is performed as follows. We start by, per method, assigning region variables to all variables (including the arguments of the method) and we assign region variables for the returned value. The number of region variables follows from the region sort of its type, as formalized in $P_\Gamma(\tau)$. However, we remove the strictness information of the argument of the method, as strict types in a function are only relevant for saturated calls and thus complicate partial applications. By removing the strictness annotation, we introduce more region arguments but we do not use them. The regions for local variables other than the arguments of the method will become additional region arguments. Some of these will be removed in the escape check or during collapsing.

We gather outlive constraints on these region variables for the first order parts of the method, that is, all but function calls and thunk allocations. We find the containment constraint for each variable and we get constraints for instructions like `let` and `match`. These constraints are explained in chapter 7.1.

Similarly, we assign one annotation variable per local variable, including the arguments of the method. We also assign one annotation variable per method, which will denote the annotation of a method. We gather annotation constraints on those variables (chapter 7.2). These constraints are used to assign additional region arguments, as described in chapter 7.2.1.

These constraints are then transformed into (possibly mutual recursive) equations on the annotation variables and annotations representing the lifetime relations that all calls in the method enforce. We solve these constraints by introducing fixpoints, which we describe in 7.3. At this point, we have an annotation in the annotation language, which we simplify with the evaluation rules.

## 7.1 Gathering outlive constraints

We generate the following outlive constraints per instruction:

**return** We generate subeffecting constraints, as the regions of the returned value must outlive the return regions.

**match** We generate constraints that the region variables of the fields of the extracted fields are the same as the related regions of the object.

**letalloc** For data type or tuple allocations, we generate containment constraints saying that the regions on the fields outlive the regions on the data type or tuple value. Thunk allocations are handled later.

**let** For the `let` instructions, we generate constraints depending on the expression. For `var`, we give constraints that the regions of the left hand side are the same as the regions of the right hand side. The `eval` expression is similar, but we must ignore the thunk region. The `castthunk` expression is similar, but here we also give constraints that the thunk region of the result is the same as its value region: the thunk region is namely not used at runtime. Note that for correctness, it would be sufficient to generate the outlive constraints in one direction, saying that the right hand side outlives the left hand side. However, we do not loose precision by generating the outlive constraints in both directions, as equality constraints. This will cause that the region variables are unified, which reduces the number of additional region arguments, which has various benefits. This removal may otherwise be handled by collapsing. By doing it now we have smaller inputs to later steps and the collapsing may not be sufficient for all cases.

For `instantiate`, we generate subeffecting constraints, that the right hand side outlives the left hand side. The instantiation may have caused that we have more region variables on the left. We would thus loose precision by generating the constraints in both directions (as equality constraints) and we thus only generate the constraints in one direction.

For the `phi` expression, we generate subeffecting constraints per branch. The call expression is handled by the annotation constraints which we introduce later. For the other expressions, we do not have to generate outlive constraints.

After computing all these constraints and the containment constraints, we must transform them into a lifetime relation. We do so by constructing the transitive closure; the algorithm we use is described in chapter 9.1.

## 7.2 Annotation constraints

To handle call expressions and thunk allocations, we generate constraints. A constraint is either an equation, saying that a variable is equal to some annotation, or it represents a call. A constraint with an equation is used to for instance join the various annotations off the branches of a phi node, or the annotations of all variables which are returned in a `return` instruction. A constraint representing a call consists of the following information:

- The kind of the constraint, which is either a direct call, thunk allocation or an instantiation.
- The variable name where the result of the call is stored. This is used to uniquely identify the constraint.

- The annotation variable of the result and its sort.
- The region variables of the result.
- The target of the call, which is either the name of a global function with its arity, or the annotation variables and regions of a local variable.
- A list of additional region arguments for this call. This list is empty for calls to local variables (higher order calls).
- A list of arguments. For type arguments, it contains the type. For value arguments, we store the annotation and regions. An instantiation may only contain type arguments.

### 7.2.1 Additional region arguments assignment

We use the call constraints we showed in the previous section, to assign more region variables. We namely need regions for the additional region arguments of calls to global functions. This will increase the number of additional region arguments of the function we are assigning. We thus need to handle self or mutual recursive functions separately. We first assign additional region arguments to the calls which are not recursive and then assign the recursive arguments.

For self recursive functions we pass the same additional region arguments on the recursive calls; the analysis is invariant over those region arguments. However, the escape check and collapsing may remove additional region argument later on, which may prevent that the region is invariant on all recursive calls.

For mutually recursive functions, we also want to assign the recursive calls to the same method the same arguments, though the recursion may be indirect via some other methods. We construct a call graph with the recursive calls. The graph is directed, as we must distinguish caller and callee, and it is a multigraph, as a method may call some method multiple times. We compute the number of additional region arguments each method will get, by traversing the graph. We start at the method whose region count we are computing, and follow all its edges. We keep a stack of the vertices which are on the current path and if we reach a vertex which is already on that path, then we stop the traversal. We have now found an (indirect) recursive call, which will be given the same region arguments and thus does not count in the region count computation. We sum the already assigned region arguments of each method for each time that we visited it in the traversal and get the number of additional region arguments per method. We use that to again traverse the graph, but this time to add the additional region arguments to each call constraint. Recursive calls are then given the same region arguments, which we can do now, as we know the number of additional region arguments.

## 7.3 Solving annotation constraints

After assigning the additional region arguments to calls, we can convert the call constraints to equations. Each equation contains an annotation variable and its sort, which form the left hand side, and an annotation, the right hand side. Each annotation variable that we introduced should have exactly one equation, except for the annotations of the arguments. We solve the equations by inlining the annotations. For loops, which may either be within a method by the usage of (conditional) jumps, or interprocedural by the usage of calls, we will obtain a recursive equation. We handle those by keeping a stack of all the variables that we have already inlined during the inlining. For each variable that we encounter, we create a fixpoint. The body of the fixpoint is the right hand side of the equation,

after inlining. We inline that right hand side with the variable, the left hand side of the equation, added to the stack. If we then recursively again need to inline that variable, we will refer back to the recursive argument of the fixpoint. This way we can generate finite annotations for methods with loops.

As many of the fixpoints are unnecessary, as the recursive argument is not used, we have an additional pass which will remove all unnecessary fixpoints. This improves the performance a little.

# 8

# Integration with compiler optimizations

## 8.1 Intermediate representations

In the Helium compiler, we use multiple intermediate representations, which operate on different levels. We start with an Haskell AST, which is then desugared to Core. The Core language is typed and we preserve that type system when going to Iridium. Having a type system gives more high level information on the program, even though the control flow of Iridium is low level. This architecture worked well for writing the region inference. Type and effects analysis are best performed on functional languages, but for region inference we saw various advantages to performing it on an imperative language. By preserving the functional type system, we got the best of both worlds.

As Iridium contains loops, we must also take those into account in the analysis. For the equations on annotations variable that we generate and solve (chapter 7.3), this implies that the equations within a method may be recursive. As we already needed to support recursive equations for recursion through calls, it did not require much work to also use that for loops within a method. That may not be the case for all analyses; for problematic analyses one may choose to perform the analysis before the tail call optimization, as that is the only part of the compiler which introduces loops. Without this pass, the control flow graph is always a directed acyclic graph.

## 8.2 Pipeline

The new backend of the Helium has various passes, which are done for correctness and performance. Details on the passes, except the region inference, are presented in [16][1].

**Haskell**

- Parsing
- Type checking
- Desugar to Core

**Core**

- Rename - Make all variable names unique
- Saturate - Make all applications saturated
- LetSort - Reduce sizes of recursive let declarations
- LetInline - Inline let declarations
- Normalize - Transform the program such that "most" subexpressions are variables
- Strictness - Promote lazy let bindings to strict bindings
- RemoveAliases - Removes aliasing of the form `let x = y in e`
- ReduceThunks - Reduces the number of thunks by making some simple expressions strict
- Lift - Moves nested function declarations to top level function declarations
- Transform to Iridium

**Iridium**

- DeadCode - Removes dead code, dead variables and dead arguments
- TailRecursion - Transforms tail recursive functions to loops
- RegionInference

## 8.3 Tail call optimization

Tail call optimization converts tail recursive functions into loops, which improves the performance and may prevent stack overflows. However, tail call optimization and region inference may influence each other negatively. First consider the case that we want to apply tail call optimization after region inference. A tail call is a recursive call at the end of a block, but the region inference analysis may have caused that a deallocation was added between the call and the end of the block. This prevents the tail call optimization. On the other hand, tail call optimization may cause that many objects are placed in the same region, whereas they would otherwise be placed in different regions. As the analysis can currently only assign regions for the lifetime of a whole method call, we cannot create regions with the scope of an iteration of the loop. Instead, we currently must assign that region for the lifetime of the whole method, which causes that objects are retained longer. A solution would be to assign regions for the lifetime of blocks or even instructions.

We thus chose to apply the tail call optimization before the region inference analysis, as the limitations that we get this way, can be solved by making the assignment of lifetimes

[1]The ThunkArity pass has been removed, as the new thunk evaluation strategy shown in chapter 8.6 made the pass redundant

more fine grained. Furthermore, the region inference operates on an imperative language, so we can also perform the inference on a program with loops as opposed to the region inference in the MLKit compiler.

In the MLKit compiler, they introduced a *storage mode analysis* [9] to overcome a similar problem. They applied the analysis on a functional language instead of an imperative language, but had similar problems with tail recursive functions. The storage mode analysis tries to reset or empty a region during its lifetime. However, this analysis was complicated to implement and was unstable over minor program changes [53, section 9]. Alternatives to the storage mode analysis have been presented in [3], [14] and [24].

## 8.4 Laziness & strictness analysis

Laziness, also known as call-by-need semantics [7], may complicate making an accurate region inference analysis. Laziness is handled at runtime by constructing an object (called a *thunk*) which represents the computation. The thunk contains (references to) the arguments of the computation. When the expression is evaluated, the thunk is replaced with (an indirection to) the result. Because of containment, the arguments of the thunk must outlive the thunk.

Consider a thunk representing the computation `length xs`. This will retain a large data structure if the list is long or the elements of the list are large, whereas the result of this computation is only one integer. Call-by-need semantics can thus cause that objects are retained longer and this can spread throughout the program. In the example, this may cause that regions containing list `xs` and its elements get a very long lifetime, as the result of `length xs` is used elsewhere in the program. We note that laziness can also reduce memory usage. The program `last [1..n]` can run in constant memory with call-by-need semantics, whereas the memory usage would be linear in $n$ when evaluated eagerly.

Garbage collection will also suffer from laziness, as this is not specific to region inference. However, this may have a larger impact on region inference, as the analysis may overestimate the lifetimes of arguments of a thunk. After evaluating a thunk, the arguments of the thunk may be released and a garbage collector can indeed clean them up. A region inferencer cannot do that by default, as the arguments of the thunk will have a lifetime which is at least as long as the thunk itself. We would need to give the thunk two lifetimes: one for the non-evaluated state and one for the evaluated state. A short lifetime for the non-evaluated state would cause that the arguments of the thunk can be released early. This will of course complicate the analysis. Furthermore, it appears that many examples which benefit from this more precise region inference, can also be handled by strictness analysis [11, 29] and some standard transformations like inlining.

To handle laziness, we thus perform a strictness analysis before the region inference analysis. The strictness analysis is performed on the Core language, such that other analyses on Core and Iridium can also benefit from strictness information. Furthermore, by analysing strictness in a different analysis, the complexity of the region inference is reduced.

The new backend of the Helium compiler did not have a strictness analysis yet. During the thesis we implemented a simple strictness analysis, which only operates on the first order parts of the program. It supports strict data fields.

## 8.5 Dead-code analysis

A dead-code analysis is performed on Iridium, which removes dead code, unused variables and unused arguments. Removing arguments will also change the type of a method and

impact partial application, so we can only remove an argument if all uses of the method pass that argument. If an argument cannot be removed because a partial application does not yet have that argument passed to it, we instead try to pass the `undefined` expression of Iridium to that argument. Note that this is an undefined value, not the Haskell `undefined` function.

Although one may expect that the removal of an argument only has a positive impact on the performance, the removal may cause that a method loses all its arguments. We then construct a global thunk for that method and its computation is shared among all uses, which will decrease the computational costs. For the region inference this may however cause that the additional region arguments of the method are also shared among the usages and thus need to be instantiated with $\rho_{global}$. This is a limitation of the runtime, as we need to pass the region arguments when constructing the thunk and thus need to share them. If the method has additional region arguments, we thus cannot conclude whether it is better to share the value, as it has both positive and negative implications.

## 8.6 Thunk evaluation & partial applications

Iridium supports multi-parameter thunks, which are used for laziness, partial applications and higher order calls. As this impacts the usage of memory and thus region inference, will discuss the thunk evaluation strategy here.

The LLVM backend previously used a different representation and evaluation of thunks, which needed to change. The previous approach had unpredictable memory usage, both in terms of the lifetimes of objects and in the sizes of allocated objects. The latter is problematic when extending the region inference analysis with multiplicity analysis, which analyses the sizes of regions. In this section, we present the new implementation of thunks which we implemented during the thesis.

Push/enter and eval/apply are two classic approaches for evaluating thunks [38]. We implemented a different evaluation strategy, which is optimized for memory behaviour regarding caches. It does so by annotating thunk objects with more information, such that we have to follow fewer pointers.

### 8.6.1 Thunk representation

A *thunk* is an object used at runtime to represent a lazy computation or a partial application. It points at a function or another thunk, as thunks may be chained as linked lists. When applying a partially applied function, we create a new thunk pointing at the previous one. A thunk may also provide multiple arguments, thus linking does not have to be used to create a list of arguments. This is done for performance, as linked lists are expensive by their unpredictable memory accesses.

We distinguish primary and secondary thunks. Thunks at the start of a chain, i.e. not pointing at another thunk, are called *primary thunks*. Thunks pointing at other thunks are *secondary thunks*. A thunk may oversaturate the function application, which means that too many arguments were passed to a function. Secondary thunks should provide at least one argument. A primary thunk pointing at a function with arity zero may not be oversaturated.

The runtime representation of a thunk consists of the following fields:

1. `header`: A header of the object, which may be used by the garbage collector. This is currently not yet used.

2. `next`: Thunks form a linked list. This field points to the next thunk. If the thunk is a primary thunk, than it will point at the thunk itself. This assures that we can always dereference this field, also in case of a primary thunk.

3. `target`: Points at the trampoline function of the applied function.
4. `value`: Contains the value of the computation after evaluating an (over)saturated thunk. Shares the same word as the *target* field, as the function pointer is not needed any more after evaluating the thunk.
5. `remaining`: The number of remaining arguments of the function call. If the field is zero, then the thunk is saturated and if the field is negative then the thunk is oversaturated. It may also contain a magic value (32766 or 32767) to denote some state.
6. `given`: The number of argument given in this thunk.
7. `arguments`: The arguments of the thunk, in reverse order. Note that the LLVM type, `[0 x {i1, i8*}]`, as seen in figure 8.6.1, is an array type of length zero. However, the real length is `given` and it is allowed in LLVM to index the array outside of its bounds, assuming that enough memory was allocated. This is the intended way to implement 'pascal style' arrays in LLVM [1, chapter Aggragate Types, Array Type]. The arguments are reversed, as the trampoline function will extract the arguments in reverse order.

### 8.6.2 Thunk state

We distinguish various states of a thunk:

1. `unsaturated` if $remaining > 0$ and $remaining < 32766$: Not enough arguments were passed. The thunk represents a partial application.
2. `saturated` if $remaining = 0$: Exactly enough arguments were passed. The thunk is not yet evaluated, but may be evaluated to weak head normal form.
3. `oversaturated_self` if $remaining < 0$ and $-remaining < given$: Too many arguments were given. The thunk is either a primary thunk or the target is not saturated.
4. `oversaturated_target` if $remaining < 0$ and $-remaining >= given$: Too many arguments were given. The thunk target of the thunk is saturated or oversaturated
5. `blackhole` if $remaining = 32767$: The thunk is being evaluated.
6. `evaluated` if $remaining = 32766$: The thunk is evaluated.

```
struct Thunk {
  i64 header;
  Thunk* next;
  union {
    Trampoline* target;
    i8* value;
  };
  i16 remaining;
  i16 given;
  [0 x {i8*, i1}] arguments;
}
```

**Figure 8.1:** The runtime representation of a thunk

If a thunk is in the state *oversaturated_target*, then the fields *remaining* and *target* may not be accurate. They may have the value of a moment back in time. Consider the following example, which is written in Haskell instead of Iridium for familiarity of the reader.

```
f :: Int -> Int -> Int -> Int
f x y z = x + y + z

a = f 1
b = f 1 2 3
c = a 2 3
```

The expression of `a` is unsaturated, as we only passed one of the three arguments. The binding of `b` is saturated; all arguments are passed. The binding of `c` is also saturated, as we passed the two remaining arguments.

A thunk can be oversaturated if the method returns a thunk representing a partial application. An example is shown below, where function `g` takes an argument and then returns another function.

```
g :: Int -> Int -> Int -> Int
g x = f x

k = g 1
l = g 1 2

m = k 2 3
n = l 3
```

As method `g` takes one argument, the binding of `k` is saturated. The value of `l` is in state `oversaturated_self`, as it passes too many arguments and is a primary thunk. The bindings of `m` and `n` are oversaturated, as they both pass 3 arguments. They are both in the state `oversaturated_target`, as their targets are saturated or oversaturated.

A thunk may also be in the state `oversaturated_self` if it is a secondary thunk and its target is unsaturated, which is the case for the value of `p`:

```
h :: Int -> Int -> Int -> Int
h x y = f x y

o = h 1
p = o 2 3
```

### 8.6.3 Trampoline functions

Thunks point at a trampoline function, which will be invoked when evaluating the thunk. The trampoline function extracts the arguments from the linked list of thunks and calls the actual function with those arguments. The trampoline function has three arguments: a pointer to the thunk, a pointer to the last argument and the number of arguments to the function in the first thunk. For a function with arity $n$, we call those parameters `thunk`, `ptr[n-1]` and `thunk_remaining[n-1]`. We represent those variables as arrays here for clarity, in the implementation they are unrolled as a list of variables. Note that `thunk_remaining[n-1]` may be less than the arity of the function, if additional arguments are passed in chained thunks, as thunks form a linked list. Furthermore, it may be less than `given`, if the thunk is oversaturated.

**Input** `thunk`, `ptr[n-1]`, `thunk_remaining[n-1]`

**Output** The result of the function call

```
next_thunk[n - 1] = thunk->next;
for (int i = n - 1; i >= 0; i++) {
  argument[i] = *ptr[i];

  to_next[i] = thunk_remaining[i] == 1;

  next_given[i] = next_thunk[i]->given;
  next_next_thunk[i] = next_thunk[i]->next;

  thunk_remaining[i - 1] =
    to_next[i] ? next_given[i] : thunk_remaining[i] - 1;
  ptr[i - 1] =
    to_next[i] ? &next_thunk[i]->arguments[0] : &ptr_i[1];
  next_thunk[i-1] =
    to_next[i] ? next_next_thunk[i] : next_thunk[i];
}
return f(arguments[0], arguments[i], ... arguments[n-1]);
```

The code is represented as a loop for clarity. In reality, the loop is fully enrolled. Arrays are replaced by multiple variables.

**Figure 8.2:** Pseudocode of a trampoline

The trampoline extracts the arguments one by one in reverse order. As shown in figure 8.2, we keep track of the number of remaining arguments in the thunk (`thunk_remaining`) and a pointer to the next thunk in the chain (`next_thunk`). When `thunk_remaining` is one, we need to go to the next thunk after extracting this argument. This can be implemented without branching by using a select instruction [1, Instruction Reference], which can be seen as a conditional expression that evaluates both branches. The loop over all arguments is fully unrolled, which makes the extraction of arguments branchless.

### 8.6.4 Evaluating a thunk

The evaluation of a thunk depends on the state of the thunk. If the thunk is unsaturated then the thunk is already in weak head normal form and we can thus return a pointer to the thunk. If the thunk is saturated, then we will write 32767 to `remaining` to denote that the thunk is being evaluated.

When evaluating a thunk in the state `oversaturated_target`, we evaluate the thunk referenced by `next` first, and update that field with the resulting value. This value is a thunk representing a partial application, as the thunk which we evaluate provides more arguments. We update `target` with `next->target` and `remaining` with `target->remaining - given`. The thunk is now in a different state and we will restart the evaluation of the thunk with the new state.

If the thunk is in the state `oversaturated_self`, we call the trampoline function starting with argument `&thunk->arguments[-thunk->remaining]`. This will yield a thunk representing a partial application. Using the number of oversaturated arguments of the initial thunk, we check whether we need more evaluation, that is, if the partial application becomes (over)saturated. If it becomes (over)saturated, then we adjust the

initial thunk by updating `next`, `target`, `remaining` and `given` and restart the evaluation procedure on the new state of the thunk. If the partial application remains unsaturated, then we will not update the existing thunk, but we will instead allocate a new thunk, containing the oversaturated arguments of the initial thunk, pointing at the thunk representing the partial application.

An alternative approach in this situation is to update the old thunk, in place. This would however also add a constraint to the region inference that a thunk should be stored in the same region as the resulting value after evaluation, which will cause that many arguments of thunks will be retained for a long time. By actually doing an additional allocation here, we may thus probably reduce the overall memory usage.

When we find a thunk in the blackhole state, we detect a loop as we are recursively trying to evaluate a thunk which we were already evaluating. We crash the program in this case. Finally, if a thunk is already evaluated, we will return the computed value stored in the `value` field.

# 9

# Lifetime relations as graphs

Lifetime relations are in the implementation represented as lists of pairs, containing all constraints of the form $\rho_1 \geq \rho_2$ including ones implied by transitivity, or as outlived-by sets. The first representation is easy to work with in the evaluation of the lambda calculus, with for instance evaluation steps with substitutions or renaming. The second representation, with outlived-by sets per region $\rho$, containing the set of regions $\rho' \neq \rho$ such that $\rho' \geq \rho$, is more performant for certain algorithms, including the join of lifetime relations, the escape check and the removal of additional region arguments. That second representation treats the relation as a directed graph, where the regions are vertices and for each constraint $\rho_1 \geq \rho_2$ of distinct region variables, we add an edge $(\rho_2, \rho_1)$, e.g. $\rho_2 \to \rho_1$. Note that we swap the positions of $\rho_1$ and $\rho_2$ when going from the relation to the graph. The successor set of a vertex $v$ is the set of vertices $u$ such that there is an edge $(v, u) \in E$, and is denoted by $N_E^+(v)$. The successor sets are the outlived-by sets of a vertex.

The relation is transitive, which implies that if there is a path from $\rho_1$ to $\rho_2$, then there is also an edge $(\rho_1, \rho_2)$. The relation is also reflexive, but we leave that information implicit and do not add self loops (edges of the form $(\rho_1, \rho_1)$)).

In the remainder of this section, we will consider the graph algorithms that we used. Furthermore, we discuss the choice of the transitive closure, which makes all outlives constraints which can be deduced using transitivity explicit, as opposed to the transitive reduction, which enforces that all relations which can be deduced using transitivity are left implicit.

## 9.1 Transitive closure

As we only defined transitivity in the context of relations, we will also need to define transitivity in the context of graphs.

**Definition 9.1.1** (Transitive graph)**.** *A graph is transitive if for each pair of vertices $u$, $v$ with a path from $u$ to $v$, there is an edge $(u, v)$.*

**Lemma 9.1.1.** *The graph representation of a lifetime relation is transitive.*

*Proof.* This follows from the definition of a lifetime relation, which requires it to be a transitive relation. □

When gathering constraints from a method, we may however start with a graph which is not transitive. To convert this into a lifetime relation, we must make this graph transitive, by adding more edges. This is formalized as a *transitive closure*.

**Definition 9.1.2** (Transitive closure)**.** *A transitive closure of a graph $G = (V, E)$, denoted by $G^T = (V, E^T)$, is the smallest*[1] *transitive graph containing edges $E$.*

Various algorithms exist to compute the transitive closure of a graph. For our implementation we used an adaption of the STACK_TC[44]. This algorithm has the following properties:

- The successor sets are computed per strongly connected component[2]
- A mapping from vertex to strongly connected component is generated
- The number of union operations, which are the most expensive operations, is reduced

We can use the mapping from vertices to strongly connected components to reduce the number of additional region arguments. The mapping functions as a substitution of region variables. However, we cannot rename region variables of an argument or a return type; we may only rename additional region arguments. A strongly connected component can only contain one region variable which is not an additional region argument, which thus is the region of an argument or the return value. We can thus either substitute all regions in a strongly connected component with the single region of an argument or the return value, or we substitute them with a new additional region argument.

The algorithm also keeps track of self loops in a strongly connected component. If such component contains multiple vertices, then it always has a self loop, otherwise we must have had an edge $(v, v)$ for a component of one vertex $v$. However, as we don't store self loops as we already know that a lifetime relation is reflexive and we thus leave that information implicit, we can remove the code which handles self loops.

Our adaption, which we call STACK_TC_REGION, is shown in figure 9.1. As the changes in our algorithm are small, we refer to [44] for an explanation and correctness proof of the algorithm.

In the loop over the successors, the original algorithm STACK_TC checks whether an edge is a so called forward edge. We do not perform this check, which may cause that a component is pushed multiple times to *cstack*. As duplicates are removed in the topological sort, these will be removed later on and not cause problems with correctness.

### 9.1.1 Cyclic outlife constraints from containment

The input of the transitive closure algorithm is generated by gather constraints over the first order parts of a function, function calls are thus ignored at this phase. Containment constraints are generated for the arguments. Those constraints may be cyclic in some conditions, which may happen with the following data type:

```
data A = A (Maybe A)
```

Let $\rho_m$ be the region of the values of `Maybe` and $\rho_a$ the region of the nested values of type `A`. For clarity, we ignore the regions for thunks. As the `Maybe` value, in case of a `Just`, contains a value of type `A` stored in region $\rho_a$, we have the constraint $\rho_a \geq \rho_m$. As that value of type `A` in region $\rho_a$ again consists a value of type `Maybe A` and the

[1] We say that graph $(V, E_1)$ is smaller than $(V, E_2)$ if $E_1 \subset E_2$

[2] A maximal set of vertices such that all contained vertices are mutually reachable

**Input** A directed graph $(V, E)$ with no self loops

**Output** The successor sets of the transitive closure minus self loops ($Succ$) and the substitution of region variables ($Comp$).

$vstack := []$
$cstack := []$
$freshRegion :=$ A fresh region variable index
**for each** $v \in V$
  **if** $v$ not already visited **then** STACK_TC_REGION($v$)

**procedure** STACK_TC_REGION($v$)
  $Root(v) := v$
  $Comp(v) := Nil$
  $push(v, vstack)$
  $SavedHeight(v) := height(cstack)$
  **for each** $w \in N_E^+(v)$ **do**
    **if** $w$ is not already visited **then** STACK_TC_REGION($w$)
    **if** $Comp(w) = Nil$ **then** $Root(v) := min(Root(v), Root(w))$
    **else** $push(Comp(w), cstack)$

  **if** $Root(v) = v$ **then**
    $vertices := []$
    Pop vertices from $vstack$ and add them in list $vertices$
    until and including we pop $v$
    **if** $vertices$ contains only additional region arguments **then**
      $C := freshRegion$
      $freshRegion := freshRegion + 1$ (the next fresh region variable)
    **else** $C :=$ the vertex of $vertices$ which is not an additional region argument
    **for each** $w \in vertices$ **do** $Comp(w) = C$
    $Succ(C) := \emptyset$
    Sort the components in $cstack$ between $SavedHeight(v)$ and
    $height(cstack)$ into a topological order and eliminate duplicates
    **while** $height(cstack) \neq SavedHeight(v)$ **do**
      $X := pop(cstack)$
      **if** $X \notin Succ(C)$ **then** $Succ(C) := Succ(C) \cup \{X\} \cup Succ(X)$

**Figure 9.1:** Transitive closure algorithm STACK_TC_REGION, modification of STACK_TC [44]

region arguments for the recursive position are instantiated the with the same regions, we have a nested field stored in region $\rho_m$, thus $\rho_m \geq \rho_a$. One solution is to modify STACK_TC_REGION to accept these cyclic constraints. This will however cause that we again generate cyclic constraints in the output and generate significantly more edges as for each edge from or to $\rho_a$, we must also construct one from or to $\rho_m$ and the other way around. An alternative solution is to apply preprocessing to the data types, to reduce the number of assigned region variables. In this case, we don't need to assign separate regions to $\rho_a$ and $\rho_m$. By only assigning one region argument for those, we reduce the number of regions and thus also reduce the number of edges in the graph representation of a lifetime relation. We thus prefer that solution, but because of time constraints we could not implement it in time.

## 9.2 Join of lifetime relations

When computing the join of annotations, we may need to compute the join of two lifetime relations. We treat this as a graph problem, where we are given two directed graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$, and need to find the smallest transitive graph containing edges $E_1 \cup E_2$.

This graph may be constructed by computing the transitive closure of $G = (V, E_1 \cup E_2)$, but that may not be the most efficient way to compute the join. In out implementation, we use an algorithm optimized for specific kinds of inputs. We expect that the domain of the edges of the first graph has little overlap with the domain of the edges of the second graph. We will thus probably need to add only few edges, even though we must add $\mathcal{O}(|V|^2)$ edges in the worst case.

For the transitive closure, it should hold that for each edge $(v, w)$,

$$N^+(v) \supset N^+(w) \tag{9.1}$$

This can be rewritten as $N^+(v) = N^+(v) \cup N^+(w)$. We use this formulation to perform a fixpoint iteration. We will however not perform this on all edges, but only on edges which may be the start of new shortest paths. This way we make use of the fact that the two input graphs are transitive and the assumption that the graphs will partly operate on different pieces of the domain. We will construct a set of possible start edges of paths, denoted by $S(E_1, E_2)$.

**Definition 9.2.1.** *Given two transitive graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$, we define the set $S(E_1, E_2)$ as the set of all edges $(v, w) \notin E_1 \cup E_2$ such that there is some vertex $u$ with $(v, u) \in E_1 \wedge (u, w) \in E_2$ or $(v, u) \in E_2 \wedge (u, w) \in E_1$.*

**Lemma 9.2.1.** *Let $v$ and $w$ be vertices in $V$, $v \neq w$ such that there is no edge $(v, w) \in E_1 \cup E_2)$. If there is a path from $v$ to $w$ in graph $(V, E_1 \cup E_2)$, then for any shortest path $v, u_2, u_3, \ldots w$ from $v$ to $w$, it holds that $(v, u_3) \in S(E_1, E_2)$.*

*Proof.* As $v \neq w$ and there is no edge $(v, w)$ in $E_1$ nor $E_2$, the path has at least two edges. The path starts with edge $(v, u_2) \in E_1 \cup E_2$. Without loss of generality, assume that $(v, u_2) \in E_1$. The lemma claims that $(u_2, u_3) \in E_2$, so assume to the contrary that $(u_2, u_3) \in E_1$. As graph $(V, E_1)$ is transitive, there is also an edge $(v, u_3)$. We can make a shorter path from $v$ to $w$ by removing vertex $u_2$ from the path and directly going from $v$ to $u_3$. Thus, the path that we started with is not a shortest path. This is a contradiction, thus $(u_2, u_3) \in E_2$ and $(v, u_3) \in S(E_1, E_2)$. □

The algorithm works by repeatedly updating the successor sets of vertices in $S(E_1, E_2)$. We write $n_i^+(v)$ for the (partial) successor set of vertex $v$ in iteration $i$. We start by initializing the all successor sets based on the union of $E_1$ and $E_2$. The algorithm is shown in figure 9.2.

**Input** Two transitive graphs $(V, E_1)$ and $(V, E_2)$

**Output** The successor sets of the transitive closure of $(V, E_1 \cup E_2)$

$$n_0^+(v) := N_{E_1}^+(v) \ \cup \ N_{E_2}^+(v) \ \cup \ N_{S(E_1,E_2)}^+(v)$$
$$i := 0$$
**do**
$$n_{i+1}^+(v) := n_i^+(v) \ \cup \ \left( \bigcup_{(v,w) \in S(E_1,E_2)} n_i^+(w) \right) \ \setminus \ \{v\}$$
$$i := i + 1$$
**while** $\exists v \in V : n_{i-1}^+(v) \neq n_i^+(v) i$
**return** $n_i$

**Figure 9.2:** Algorithm for transitive closure of union

## 9.3 Escape check

The escape check tries to remove additional region arguments if they can be allocated within the analysed function. We may remove a set of region variables $D$ if for all regions $\rho_1 \in D$, there is no region $\rho_2 \notin D$ with the constraint $\rho_1 \geq \rho_2$. When converting this to a directed graph, the escape check becomes a reachability check: which regions are not reachable from the regions of the arguments and return value?

For region variables used in higher order calls, we do not know whether they escape. If they are used in an application in the annotation with lifetime context local bottom, the region will not occur on the left hand side of an outlives constraint, so we may still allocate that region within the function (if there are no other constraints on that region). For other applications, we cannot conclude that the region variable does not escape.

## 9.4 Cycle detection

If we find an additional region argument $\rho_1$ which has the constraints $\rho_1 \geq \rho_2$ and $\rho_2 \geq \rho_1$, we may substitute $\rho_1$ with $\rho_2$. For determinism, we require that if $\rho_2$ is also an additional region argument, then it must be defined later in the lambda introducing the additional region arguments. If that is not the case, then the substitution should be applied with the roles of $\rho_1$ and $\rho_2$ swapped.

Cycle detection is performed on a normalized annotation. From this annotation, we extract the lifetime relation which corresponds to the fully saturated call, with all annotation arguments substituted with bottom. This lifetime relation is converted to a graph. As we work with transitive graphs, cycles can be detected easily. We traverse the edges and for each edge $(\rho_u, \rho_v) \in E$, we check whether there is an edge $(\rho_v, \rho_u)$.

The cycle detection produces a substitution map and a new graph. The map contains the substitutions of the additional region arguments. The graph is the input graph with all cycles removed. For each cycle, we replace all regions on the cycle with the region with the lowest index, to achieve determinism. This graph thus also has cycles removed of regions which are not additional region arguments. This is needed as collapsing, which we discuss in the next section, operates on directed acyclic graphs.

## 9.5 Collapsing

With collapsing, we remove additional region arguments if we won't lose precision by doing so. Collapsing is defined based on outlive sets in chapter 5.6.3. Outlive sets correspond to predecessor sets (denoted by $N_E^-$) instead of successor sets. We perform a depth first search (DFS) in reverse direction (by following the predecessors).

For each vertex $u$, we first check whether it may be substituted. This can only be done if $u$ is an additional region argument and if $u$ is not used in applications, except ones with lifetime context $\perp$, local bottom. We check whether there exists a predecessor $v$ such that $N_E^-(u) = N_E^-(v) \cup \{v\}$. If such vertex $v$ exists, it is unique as collapsing operates on an acyclic graph.

When designing this algorithm, we expected that the depth first search would cause that we only needed a single traversal over the graph. However, that is not the case and we thus currently perform multiple iterations of this procedure until a fixpoint is reached.

## 9.6 Transitive reduction

In our implementation we chose a representation of lifetime relations as a transitive graph, which makes all constraints implied by transitivity explicit. The transitive reduction [2] is an alternative, which enforces that all constraints which can be deduced by transitivity are left out.

**Definition 9.6.1** (Transitive reduction)**.** *The transitive reduction of graph $G$, denoted by $G^t$, is the smallest graph $H$ such that $H^T = G^T$.*

The proof of uniqueness can be found in [2]. A representation based on the transitive reduction would have the clear advantage that the structure is more compact; we store fewer edges. However, this representation is computationally more expensive. Algorithms [2] to compute a transitive reduction will usually do that by first constructing a transitive closure, so we will clearly require more time and it may be harder to compute the transitive reduction incrementally, for instance when computing the transitive reduction of two graphs which are a transitive reduction. Furthermore, reachability checks are more expensive, as we only need to check the existence of a single edge if we have a transitive closure.

The choice of representation thus clearly brings a trade off between time and memory usage. We chose in favour of the transitive closure representation, as the disadvantage of increased memory usage can be mitigated in various ways. First, we will unify additional region arguments, to reduce the number of vertices. As the number of edges is bound from above by $n^2$, a reduction in the number of vertices will have a large reduction in the number of possible edges. Second, we perform the analysis per binding group. We will thus usually only need to store a few graphs for lifetime relations. The graphs during the analysis may be larger than the resulting annotation, as intermediate lifetime relations may not have had additional region arguments unified. We thus may require some more memory space for intermediate values in the analysis.

# 10

# Related work

## 10.1 Region optimizations

We quickly describe some other optimizations for region based memory management. One limitation of regions is that they are lexically scoped. This is needed as they can be allocated on the stack. *Storage mode analysis* allows a region to be reset during its lifetime [9]. An alternative analysis allows non-lexical lifetimes [3]. Such regions can however not be allocated on the stack.

The region analysis adds region parameters to functions. These parameters are used at runtime to allocate objects in the specified regions. However, regions from which the function only reads, are not needed at runtime. These are called *get regions* and may be removed from the function signature [9], reducing the runtime overhead of regions slightly.

## 10.2 Region waste

In experiments [21] Tofte e.a. found that the main inefficiencies appeared to be caused by infinite regions, for which the analysis could not deduce a maximal size. Those are handled by allocating blocks on the heap instead of on the stack, which are linked together as a linked list. Many of those infinite regions actually had only a few values, meaning that a large part of such block was not used, which they call *region waste*. In their benchmark programs the region waste was approximately 20% [1].

To reduce memory leaks, region inference can be combined with a garbage collector based on Cheney's stop and copy algorithm [13]. To be able to do so, the analysis needed to be changed [18, 21]. For the garbage collector, it is necessary that there are no dangling pointers, references to memory which has already been released. This can happen with region inference, for instance when the inference finds that a list is only used to compute its length. The elements of the list could then be released. However, this causes dangling pointers and we thus need to make the analysis less precise. This is done by enforcing that values in some object have a longer lifetime than the object itself. This had only little effect on the performance on a range of benchmark programs [18].

[1] In their benchmarks, several additional optimizations were added. They used a garbage collector at runtime to reduce memory leaks for region-unfriendly programs.

## 10.3 Region inference in Haskell

JHC[2], a compiler for Haskell, has implemented region inference. Details on their region analysis lack and the work is not reported in an academic format. The website does note that the region based system leaks memory and later versions of the compiler thus combine region inference with a garbage collector at runtime.

## 10.4 As Static As Possible memory management

Proust [46] presented a memory management scheme, called As Static As Possible or ASAP, which is also based on static analysis. Based on the analysis, deallocation instructions are added to the program. Their approach has less influence on the user, as the user of a region based system must be aware of regions to prevent writing region-unfriendly programs. Their system however has larger runtime costs. Region based memory management can deallocate a region in constant time, independent of the region size. ASAP however must perform some scanning after deallocations.

## 10.5 Garbage collectors

Garbage collectors trace the heap at runtime for garbage. Classical algorithms include the mark-and-sweep algorithm, which marks all live objects in the heap using depth first traversal and then deallocates the non-live objects [40] and Cheney's copying collector, which copies all live objects during a traversal [13]. Generational garbage collectors are based on the notion of *infant mortality*: "most objects live a very short time, while a small percentage live much longer" [56] or "young objects are more likely to die than old objects" [8]. The heap is divided into several generations. One generation is used for new allocations. When that generation is full, the garbage collector traces that chapter and surviving objects are copied to the next generation. This prevents that the whole heap is scanned each time the garbage collector runs. Since objects are copied during a collection, the runtime needs to keep track of all pointers from an older generation to a younger generation, which adds minor cost to write operations [4, 31, 8].

The Glasgow Haskell compiler (GHC) [32] also uses generation garbage collection. As objects are immutable in Haskell, update operations (writes after the construction of an object) are rare. They only occur when writing the result after evaluating a thunk. This causes that there are few pointers from an older generation to a young generation, which makes the generational garbage collector in GHC efficient [47].

## 10.6 Escape analysis

Escape analysis provides an alternative approach to using stack allocations. The analysis finds object allocations which do not escape the scope in which they are created. These objects will be allocated on the stack instead of the heap. One may have advantages as better cache usage or less heap fragmentation. When used as an addition to a garbage collector, it will also relieve the garbage collector from tracking these objects. Escape analysis has been applied to both functional [10] and imperative languages [19]. Region inference can be seen as an extension to escape analysis: the latter is restricted to allocate in the current stack frame, whereas region inference can also allocate in previous stack frames.

[2] `http://repetae.net/computer/jhc/jhc.shtml`, accessed 2019-08-10

## 10.7 Side effects per region

Lippmeier [36] presented an analysis on side effects. The analysis uses regions to group objects, and analyse per region whether side effects were performed on that region. Regions are thus not used for garbage collection, but only to give structure to another analysis.

# 11

# Conclusion

In the introduction, we presented three research questions. First, we asked whether region inference can be made higher ranked. The analysis that we designed and implemented shows that this can be done. However, the analysis is constrained by the possibilities that we have at runtime. The additional region arguments form the main restriction, as they are shared among all calls to a partially applied function. Furthermore, the analysis is limited by the lack of closed forms of fixpoints. We presented a possible solution for that, by preserving a fixpoint until we have enough information to evaluate it, but we did not succeed in finding a proper evaluation strategy such that all fixpoint can eventually be computed.

The second research question regarded the adaption of higher ranked type and effect systems to a real programming language like Haskell. We have seen that this significantly complicates the analysis. The inclusion of polymorphism had large impacts on the design of the annotation language. It caused that tuples were added to the sorts of annotations and region variables, which are used to build tree structures. This is needed to handle that types may have a different number of region and annotation variables. The inclusion of polymorphism also implies that fixpoints cannot always be computed, as the number of iterations that we need to perform may depend on the instantiation of a type variable and become arbitrarily large.

Data types also complicate the analysis. They have a rather large design space. Annotations on data types remains as future work.

The third and final question regarded the interaction of region inference with other compiler optimizations. The architecture of Helium, with multi level intermediate representations, worked out really well for these optimizations. Iridium has imperative control flow and a functional type system, which allows us to perform tail call optimizations before the region inference. By applying strictness analysis before the region inference, we have to do less reasoning on laziness in the region inference analysis. However, the optimization passes influence each other negatively on some minor cases. Especially loops constructed by tail call optimization may need some more optimizations, for instance by assigning finer grained lifetimes as a region currently always have the lifetime of the full function.

## 11.1 Future work

### 11.1.1 Soundness proofs

Given the scope of the thesis project, no soundness proof could be made. For memory management, which is a fundamental part of a language, it would be desired to have a soundness proof, which says that objects will not be deallocated too early.

### 11.1.2 Multiplicity analysis

In the MLKit compiler, the region inference was combined with multiplicity analysis which tries to find an upper bound on the sizes of regions [55]. This allows regions to be stack allocated, instead of heap allocated. As most regions only contain a few objects, this greatly reduces the overhead of region based memory management. However, even with their multiplicity analysis, many heap allocated regions still had only one or a few objects. A more accurate analysis may reduce the overhead even further. The initial plan for this thesis was to improve the multiplicity analysis with techniques from resource analysis, which uses amortization to find bounds on resources like time or memory. Amortization is an important technique in (time) complexity analysis [50], which allows to average the complexity of operations over time. It was proposed as a manual way of analyzing the running time of a program, but can also be used for other resources like memory and forms the basis of automatic resource analysis [33]. It has been applied to derive linear bounds on first order functional programs [28] and was later extended to higher order programs [33], univariate polynomial bounds [27], multivariate polynomial bounds [25], arbitrary tree-like data structures [26], lazy evaluation [49] and side effects [12].

It would be an interesting question to see whether region based memory management can benefit from inferred polynomial bounds on region sizes, compared to constant bounds. The existing multiplicity analysis can namely only infer constant bounds and does not use amortization. Applying the techniques from resource analysis and adapting their analysis to give bounds per region may give polynomial bounds, which would ideally be used for code generation, but one may also use those to devise a tool that gives the user insights in the memory usage of regions, and the regions which may leak memory.

### 11.1.3 Annotations on data types

The implementation of the region inference analysis does not support annotations on data types. Given the large design space of data types, it will probably not be possible to support all data types and we will thus need to fall back to a top annotation, which enforces that objects are allocated in the global region, for data types which cannot be handled. Simple data types which only use sum and product types can be handled by concatenating the annotations of all fields in a tuple. Recursive data types can be handled by instantiating the annotations on the recursive positions with the same values, similarly as we do with the region arguments of data types. However, if the data type is recursive via a function, things will quickly become complicated. First, consider a data type which is recursive on the right hand side of a function arrow:

```
data A = A (X -> A) (Int -> Int)
```

The data type contains two functions. It is recursive via the first function. If we instantiate the annotations of the recursive position the same, we cannot use the region arguments and annotations arguments of data type `X` in the recursive positions. A solution would be to add a lambda consuming the annotation arguments of `A` in front of each annotation on this data type. In general we will pass $\perp$ to this argument, but when we

apply the first function we will pass the argument to the annotations of the recursive position. If we do this repeatedly, we take the join of all those annotations. It becomes more complicated however, if the data type is recursive on the right hand side, which is called a contravariant recursive data type.

```
data B = B (B -> X)
```

This data type gets one annotation argument, as it contains one function. The naive approach would cause that a recursive sort is assigned to this annotation argument, namely $s = [(s); (P, P, (P))] \to [(); P] \to [(); (P, ())]_{\uparrow} \to R$. However, we do not allow recursive sorts, as we would lose strong normalization. An alternative approach uses the fixpoint combinator which was added to the language. The idea is as follows: the annotation of data type `B` gets sort $[s; ()] \to s$, where $s = [(); (P, P, (P))] \to [(); P] \to [(); (P, ())]_{\uparrow} \to R$. Thus, $s$ is the same as the naive sort, with the recursive reference to $s$ removed. To compute the effects of a function extracted from the data type, we compute the fixpoint of that annotation.

Taking the fixpoint assumes that the function in `Bar` is applied with an object with the same annotation. If that is not the case, such as in function `f` below, we first take the union of the two annotations and then compute the fixpoint. This order is essential: if we take the union of the fixpoints the analysis becomes unsound.

Haskell also allows to instantiate the type arguments of a data type in recursive position differently. This means that we cannot assign the same annotation to recursive position, as the recursive position expects a different sort.

```
f :: B -> (B -> X) -> X
f b g = g b
```

### 11.1.4 Code generation

The implementation in the Helium compiler currently lacks code generation for regions. The region annotations contain the information which the code generation should use.

The program must be transformed to add allocations and deallocations of region and each object allocation must be annotated with a region variable. The latter can be done with the mapping from variables to region variables, with which the analysis starts, and the resulting region substitution of the analysis. The escape check finds which regions do not escape out of a method and may thus be allocated within that method.

Furthermore, we must pass regions as arguments to methods, for the additional region arguments and the regions for the return value. Note that the regions of arguments of a function do not require a runtime representation. As regions may form a tree structure, this must also be handled at runtime. Especially as trees may also be instantiated with only a single region argument, it requires some design to represent those trees at runtime. Note that we must allow to pass a single region to a tree as additional region arguments cannot be polymorphic. They can be represented using pointers, but that would increase the memory costs whereas we are actually trying to reduce memory costs using region based memory management. A cache-aware implementation with local pointers, which may be stored in a limited number of bits, may be a solution to this problem.

### 11.1.5 Fixpoints

The current evaluation strategy of fixpoints is not sufficient for all functions. For higher order functions which repeatedly apply a function argument on its own result, we may not always reach a fixpoint. Research into different evaluation strategies would help making the region inference practical. If no such strategy exists, widening would be a solution but

this has to be done in a consistent way such that mutual recursive functions are analysed properly.

### 11.1.6 Performance

The region inference analysis currently takes a large portion of the compile time. Compiling the Prelude currently takes two minutes, whereas it used to take under a minute. There are however some opportunities to either optimize the lambda calculus part of the analysis, as the evaluation may be made more efficient, or to improve the graph algorithms, which may have variants with lower complexity.

### 11.1.7 Lifetimes within methods

Regions currently get the lifetime of a method. Especially with loops generated by tail recursion optimization, it would be beneficial to either clear a region during its lifetime or assign shorter lifetimes to regions. For instance, by letting blocks or instructions be the units on which lifetimes are assigned. This may either be done by adapting the region inference algorithm or as a post-processing step, which performs small changes to reduce the lifetimes.

# Bibliography


[1] Llvm language reference manual. *Available online:* `http://releases.llvm.org/8.0.1/docs/LangRef.html`. Accessed: 2019-08-08.

[2] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[3] Alexander Aiken, Manuel Fähndrich, and Raph Levien. *Better static memory management: Improving region-based analysis of higher-order languages*, volume 30. ACM, 1995.

[4] Andrew W Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1989.

[5] Andrew W Appel. Ssa is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

[6] Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2006.

[7] Zena M Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of functional programming*, 7(3):265–301, 1997.

[8] Henry G Baker. Infant mortality and generational garbage collection. *ACM Sigplan Notices*, 28(4):55–57, 1993.

[9] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM, 1996.

[10] Bruno Blanchet. Escape analysis: correctness proof, implementation and experimental results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–37. ACM, 1998.

[11] Geoffrey L Burn, Chris Hankin, and Samson Abramsky. Strictness analysis for higher-order functions. *Science of computer programming*, 7:249–278, 1986.

[12] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. *ACM SIGPLAN Notices*, 50(6):467–478, 2015.

[13] Chris J Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[14] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275. ACM, 1999.

[15] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[16] Ivo Gabe de Wolff. The helium haskell compiler and its new llvm backend. *EuroLLVM 2019, available online* `https://youtu.be/x6CBks1paF8`. Accessed: 2019-08-06.

[17] Edsger W Dijkstra. Recursive programming. *Numerische Mathematik*, 2(1):312–318, 1960.

[18] Martin Elsman. Garbage collection safety for region-based memory management. *ACM SIGPLAN Notices*, 38(3):123–134, 2003.

[19] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction*, pages 82–93. Springer, 2000.

[20] Jean-Yves Girard. The system f of variable types, fifteen years later. *Theoretical computer science*, 45:159–192, 1986.

[21] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. *ACM SIGPLAN Notices*, 37(5):141–152, 2002.

[22] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 62–71. ACM, 2003.

[23] Bastiaan J Heeren. *Top quality type error messages*. Utrecht University, 2005.

[24] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 175–186. ACM, 2001.

[25] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(3):14, 2012.

[26] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for ocaml. In *ACM SIGPLAN Notices*, volume 52, pages 359–373. ACM, 2017.

[27] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In *European Symposium on Programming*, pages 287–306. Springer, 2010.

[28] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *ACM SIGPLAN Notices*, volume 38, pages 185–197. ACM, 2003.

[29] Stefan Holdermans and Jurriaan Hage. Making stricterness more relevant. *Higher-Order and Symbolic Computation*, 23(3):315–335, 2010.

[30] Stefan Holdermans and Jurriaan Hage. Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In *ACM Sigplan Notices*, volume 45, pages 63–74. ACM, 2010.

[31] Richard Jones and Rafael D Lins. Garbage collection: algorithms for automatic dynamic memory management. 1996.

[32] Simon LP Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.

[33] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *ACM Sigplan Notices*, volume 45, pages 223–236. ACM, 2010.

[34] Ruud Koot. Higher-ranked exception types. *Available online,* `http://www.cs.uu.nl/docs/vakken/apa/12ruud-higherrankedexceptiontypes.pdf` *and* `https://github.com/ruudkoot/phd`, 2015.

[35] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[36] Ben Lippmeier et al. *Type inference and optimisation for an impure world.* Australian National University, 2010.

[37] Simon Marlow et al. Haskell 2010 language report. *Available online* `https://www.haskell.org/definition/haskell2010.pdf`, 2010.

[38] Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In *ACM SIGPLAN Notices*, volume 39, pages 4–15. ACM, 2004.

[39] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[40] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. 1959.

[41] Sally A McKee. Reflections on the memory wall. In *Conf. Computing Frontiers*, page 162, 2004.

[42] James S Miller and Guillermo J Rozas. Garbage collection is fast, but a stack is faster. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1994.

[43] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Type and effect systems. In *Principles of Program Analysis*, pages 283–363. Springer, 1999.

[44] Esko Nuutila. Efficient transitive closure computation in large digraphs. 1998.

[45] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17:1–82, January 2005.

[46] Raphaël L Proust. Asap: As static as possible memory management. Technical report, University of Cambridge, Computer Laboratory, 2017.

[47] Patrick M Sansom and Simon LP Jones. Generational garbage collection for haskell. In *FPCA*, volume 93, pages 106–116. Citeseer, 1993.

[48] Gunther Schmidt. *Relational mathematics*, volume 132. Cambridge University Press, 2011.

[49] Hugo Simões, Pedro Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *ACM SIGPLAN Notices*, volume 47, pages 165–176. ACM, 2012.

[50] Robert E Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[51] Fabian K Thorand. Higher-ranked polymorphism in dependency analyses. Master's thesis, 2017. *Available online,* `https://dspace.library.uu.nl/handle/1874/351887`.

[52] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):724–767, 1998.

[53] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004.

[54] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value $\lambda$-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 188–201. ACM, 1994.

[55] Magnus Vejlstrup. Multiplicity inference. *Master's thesis, Dept. of Computer Science, Univ. of Copenhagen*, 1994.

[56] Paul R Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, pages 1–42. Springer, 1992.

[57] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.