Master Thesis

Higher Ranked Region Bound Inference For Region Based Memory Management In Helium

Hanno Ottens - h.p.ottens@students.uu.nl

Supervisors
MSc. I.G. de Wolff
dr. J. Hage

August 9, 2021

Abstract

Region based memory management, a compile-time alternative to garbage collection, splits the heap into a stack of lexically scoped regions. Most of the work for region based memory management can be done at compile time by inferring the placement and bounds of regions. A region bound represents the size of a region. Regions with finite bounds, can be stack allocated, otherwise they must be heap allocated.

In this thesis we design a higher ranked region bound analysis to infer the bounds of regions. The focus of this thesis lies on the implementation within the Helium Haskell compiler, opposed to proving metatheory. We show that the analysis infers a high percentage of finite bounds whilst having minimal impact on compilation time.

Contents

Contents

1 Introduction

Before we get into the details of the analysis, we will cover some of the background material that will aid in the understanding of the analysis. In this introductory chapter we will look at memory management systems and how a region based approach would work.

1.1 Memory

Programs require computer memory to run. Programs allocate memory by claiming a part of the available system memory. Program memory is often divided into two sections: the stack and the heap.

1.1.1 The stack

The stack is a fixed-size chunk of memory. The stack is built up out of stack frames. A *stack pointer* points to the end of the stack. For each function call we push a new stack frame. A stack frame contains data that is local to the function, i.e. (pointers to) parameters and/or local variables. Whenever we exit a function we pop a stack frame by moving the stack pointer back to the end of the previous stack frame. Because the stack frames are ordered in the order that we call functions, we say the stack is *lexicographically ordered*. The memory for the stack is allocated when the program is initialized, the size of the stack cannot be extended. Stack allocation is preferred because there is less memory fragmentation (everything is stored in a continuous memory frame) and better cache locality (the stack behaves predictably) [1].

1.1.2 The heap

The heap is an extendable chunk of memory. This means we can claim more system memory to extend the heap whenever we need. The heap is used for data that either cannot be stored on the stack or that we do not want to store on the stack. Objects can be allocated anywhere we want in the heap, we say the heap is *unordered*. Objects can also be deallocated in any order, which may cause memory fragmentation. Fragmented memory has unreserved bytes between allocated bytes of memory. Fragmented memory is not one continuous block of allocated memory.

1.2 Memory management

Computers do not have unlimited memory. To reduce the memory usage of programs we can deallocate parts of the memory that are no longer used. In this section we will explore two methods for deallocating memory at runtime.

1.2.1 Garbage Collection

Many modern compilers use Garbage Collection (GC). The garbage collector scans the reserved memory for objects that will not be used in the future. When it finds such an object it can be deallocated. The memory of such objects can be used for new allocations.

A popular GC technique is called generational garbage collection (GGC) [2]. Objects that are alive for a long time are likely to live even longer, while new objects are more likely to be deallocated soon. The runtime uses this premise by grouping objects by age in so-called generations. Objects that have survived for a certain amount of time are moved into an older generation. The older the generation, the less often those objects are checked for deallocation.

The garbage collector is part of the runtime and runs as a process parallel to a program. This process uses computation time to decide what memory can be deallocated, which could otherwise be used to execute program code [3].

1.2.2 Region based memory

An alternative to garbage collection is region based memory management (RBMM). RBMM splits the heap into a lexicographic stack of regions. Opposed to garbage collection, RBMM is done at compile time with region constructs either explicitly written by the user in the program code or automatically inferred by the compiler. This means that for some programs there will be no need for a garbage collection process at runtime, which in turn means that more computation time is left for running the program code. To make RBMM explicit, we introduce the following constructs.

$$\begin{array}{l} \textbf{letregion } \rho \textbf{ in } e \\ e \textbf{ at } \rho \end{array}$$

The first expression allows a region variable ρ to be used in the expression e. It allocates the space required for the region on the stack or the heap. The second expression puts the value of e into the region ρ. It puts the bytes of in the object in the allocated space of the region.

It is possible to derive an upper bound on the number of allocations. If the bound is finite the region is called *bounded*. If we are unable to derive a finite bound on the number of allocations we call the region *unbounded*. In some cases this is due to inaccuracy in the analysis, in other cases the number of allocations to a region might be unknown or truly infinite, e.g. a region that is repeatedly allocated to in infinite recursion.

Stack allocation

We can allocate bounded regions on the stack. Memory in a stack frame has to be reserved in advance, which is why an upper bound on the region must be known beforehand. Stack space is limited, thus it might still preferred to allocate on the heap if the bound is large. As mentioned before, the region bound is only an upper bound, a lot of stack space could be wasted if we allocate a region with a large bound on the stack.

Heap allocation

Unbounded regions can always grow out of their allotted space when allocated on the stack. This means that a region of unbounded size must be allocated on the heap. A representation of heap allocated regions can be found in Figure 1.1.

Unbounded regions are allocated in fixed-size *region pages*. Each unbounded region is represented as a linked list of region pages. If a region page is full another region page is appended to the back of the linked list. A region allocated on the heap has a pointer to the start of the list, the first free position and the end of the last region page.

When a region is deallocated, its region pages are appended to the end of the *free list*. The free list is a doubly linked list of the deallocated pages on the heap. Deallocation can be done in constant time, because the linked list of the to-be deallocated region can simply be appended to the end of the free list. If a new region page is needed, it can be allocated from the free list.

Unbounded regions cause issues for RBMM. These heap allocated regions cause memory fragmentation and there is no good answer to what the size of a region page should be [4]. For this reason RBMM could be combined with GC, where GC handles the unbounded regions and RBMM handles the bounded regions.

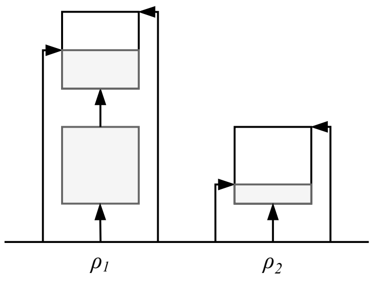

Figure 1.1: Unbounded regions on the heap

Region bounds

The bound of the region decides where the value is stored. As discussed before, putting a finite region on the stack is better than putting it on the heap. More finite bounds can make the program run faster (due to cache locality) and wastes less memory (see *region waste* in Section 3.2.1). This is why finding bounds on the size of a region is important. Inferring these bounds is the focus of this thesis.

2 Background

In this chapter we will cover some of the theoretical background required to understand the analysis. We will cover lambda calculi, lattices, fixpoints and type & effect systems.

2.1 The lambda calculus

The lambda calculus is a Turing-complete set of expressions. The basic lambda calculus can be seen in Figure 2.1. This simple language is able to encode many other features by combining expressions. Booleans and the if-then-else expression can be encoded using lambdas as can be seen in Equation 2.1. The definition of true returns the first argument, false returns the second argument and the if-then-else applies its branches to the boolean parameter, which picks the first if it is true and the second if it is false. Any abstract datatype can be encoded in such a manner, this is called the Church encoding of that datatype.

$$\textit{true} \sim \lambda x.\lambda y.x \qquad false \sim \lambda x.\lambda y.y \qquad \textit{if-then-else} \sim \lambda x.\lambda y.\lambda z.x\ y\ z \tag{2.1}$$

We define a lambda-calculus because it is easier to reason with than a more complex programming language. Many languages can also be seen as an extension of the lambda calculus. A lambda calculus can be used to prove properties of programs and define analyses. These proofs can then be transferred over to an actual programming language. The proofs will still hold on that programming language as long as the constructs of such a language can be expressed in the lambda calculus, like the encoding of the if-then-else.

Even though a language with only lambdas is already Turing complete, most analyses are designed on a larger set of expressions in the lambda calculus or with other constructs like datatypes. This allows one to handle each expression/construct in a distinct way. One might want to handle application in a different manner than an if-then-else. Adding more constructs to a lambda calculus allows for a more precise analysis.

$$\begin{aligned} e ::=\ & x && \textit{(Variable)} \\ \mid\ & \lambda x.e && \textit{(Abstraction)} \\ \mid\ & e_1\ e_2 && \textit{(Application)} \end{aligned}$$

Figure 2.1: A simple lambda calculus

2.2 Lattices

A lattice $\mathcal{L} = (L, \sqsubseteq)$ is defined as a set L with a partial order '$\sqsubseteq$'. An example of a lattice is the subset relation over the powerset of I. Take $I = \{1, 2, 3\}$, which gives us 8 possible subsets. If we would draw the lattice of $\mathcal{L} = (P(I), \subseteq)$ as a so-called Hasse diagram we would get Figure 2.2. We can see that for any two elements that are connected with a line that the subset relation holds bottom-to-top. The relation is transitive, which means the relation also holds for any bottom-to-top *path* in the diagram. There are many definitions for lattices, we choose to follow the definitions from the Davey & Priestley book [5].

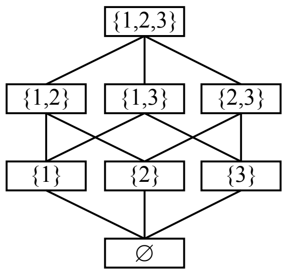

Figure 2.2: Hasse diagram of $\mathcal{L} = (P(I), \subseteq)$

2.2.1 Lower bound & upper bound

The lower bound set of some element $x \in L$ are the elements $y \in L$ for which $x \sqsubseteq y$ holds. The upper bound set of some element $x \in L$ are the elements $y \in L$ for which $y \sqsupseteq x$ holds. Looking at our subset example the upper bound set of $\{1\}$ is $\{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$ and the lower bound set of $\{1\}$ is $\{\emptyset, \{1\}\}$.

The greatest element of some set S is the element $z \in S$ for which $y \sqsupseteq z$ holds for all elements $y \in S$. The least element of some set S is the element $z \in S$ for which $y \sqsubseteq z$ holds for all elements $y \in S$. Note that a greatest element or least element does not always exist.

2.2.2 Join and meet

On a lattice we can define the join '$\sqcup$' and meet '$\sqcap$'. For elements $x, y \in L$ the join of x and y, denoted by $x \sqcup y$, is the least element of the intersection of the upper bound sets of x and y. The meet is the greatest element of the intersection of the lower bound sets. In our running example we have that $\{1\} \sqcup \{2\} = \{1, 2\}$ and $\{1\} \sqcap \{2\} = \emptyset$. Note that in this example the join is actually the union operator ($\sqcup = \cup$) and the meet is the intersection operator ($\sqcap = \cap$), but this is not always the case.

2.2.3 Bounded lattice

A bounded lattice has a single top element ($\top$) and a single bottom element ($\bot$) for which the following must hold:

(i) For all $x \in L$ we have that $\top \sqcup x = \top$.

(ii) For all $x \in L$ we have that $\bot \sqcap x = \bot$.

In the subset-lattice example, top is $\{1, 2, 3\}$ and bottom is $\emptyset$.

2.2.4 Fixpoints & iteration

On a lattice $\mathcal{L} = (L, \sqsubseteq)$, we can define a function f that given an element of L returns another element of L. The function f is called *monotone* if $x \sqsubseteq y$ implies $f\ x \sqsubseteq f\ y$ for all $x, y \in L$. If for a function f we have $f\ x = x$, then x is called a *fixpoint.* A fixpoint does not change no matter how many times we apply f.

Repeatedly applying monotone function $f : L \to L$ to its own result is called *fixpoint iteration.* In Equation 2.2 we can see that we reach a fixpoint after n steps. We use f^n to denote f applied to its own output n times. The number of steps n is not finite for all lattices. In other words: fixpoint iteration does not terminate on all lattices.

$$f^n\ x_0 = f^{n+1}\ x_0 \tag{2.2}$$

Ascending chain condition

A useful property for analyses is the ascending chain condition (ACC). The ACC assures that fixed point iteration terminates within a finite number of iterations. In Equation 2.2 the ACC would assure that n is finite.

Having a top element is not enough to make sure fix point iteration terminates. Fixpoint iteration could occur on an infinite path towards top. Take for instance the set $\mathbb{N} \cup \infty$, where $\top = \infty$. If we would start at any number and our fixpoint iteration would increase said number by one each iteration, we would never reach top.

Any lattice of finite size is guaranteed to have an ascending chain condition, because we can only do as many fixpoint iterations as the longest bottom-to-top path in a lattice. We know the lattice is finite, which means the longest bottom-to-top path must also be finite.

2.3 Type & effect systems

Types are properties of values, for instance the number 0 is of type *int*, because it *is* an integer. Besides the types of a program we can also infer a so-called *effect.* Effects are properties of evaluation, for example with call tracking analysis (CTA) we want to know the set of lambdas which *may be* called during evaluation. We give all lambdas a unique label, for example λ_F, such that we can identity them. The effect of '$(\lambda_F x.x+1)\ 2$' would be '$\{F\}$', because the lambda labeled F is called during evaluation. The effect is often denoted as an annotation language on the types.

Effects can for example be denoted as sets of atomic effects. The symbol π in Figure 2.3 denotes an atomic effect, ϕ and ψ are sets of effects. The effect annotations can be put wherever they are needed, for instance on the function arrow or on the base types. An effect on an arrow indicates that the effect may occur when the function is applied.

$$\begin{aligned} \phi, \psi &::= \{\pi\} \mid \phi \cup \psi \mid \emptyset \\ \hat{\tau} &::= \mathit{int}\langle\psi\rangle \mid \mathit{bool}\langle\psi\rangle \mid\ \ldots\ \mid \hat{\tau}_1 \xrightarrow{\psi} \hat{\tau}_2 \end{aligned}$$

Figure 2.3: A simple type annotation language

In the case of CTA the atomic effect would be a unique identifier for a lambda (e.g. 'F'). The effects would then only be placed on the arrow, as we are only interested in which functions are executed.

Type & effect analyses are often an approximation. The input of a program is unknown at compile-time, which means that we do not know which program path an execution will take. It is also impossible to inspect all program paths, as there might be an infinite number of them. This means that we might have to overestimate the effects. It is important that the approximation is sound. We will often have to resort to a suboptimal result to make sure that the result is sound. For a more complete overview on type & effect systems we recommend reading the book by Nielson & Nielson [6].

2.3.1 Poisoning

A common problem in type and effect systems is called *poisoning.* Poisoning occurs when the usage of a variable in one place affects its annotated type for other usages of that variable. Lets take a look at how this could happen in CTA for the program in Figure 2.4.

$$\begin{aligned}
\textbf{let } & f = \lambda_F x.\ x + 1 \\
\textbf{in let } & g = \lambda_G y.\ y * 2 \\
\textbf{in let } & h = \lambda_H z.\ \textbf{if } z = 0 \textbf{ then } f \textbf{ else } g \\
\textbf{in } & f\ 1
\end{aligned}$$

Figure 2.4: Poisoning of f

In a naïve implementation, f would need to have the same type at all of its usages. First we see that f and g should at least have $\{F\}$ and $\{G\}$ on the arrow respectively, because those lambdas are called when evaluating that function. So we guess that the types of f and g are '$Int \xrightarrow{\{F\}} Int$' and '$Int \xrightarrow{\{G\}} Int$' respectively.

In the branches of the if-then-else the types of f and g must be the same, however they are not. So we change our guess of the types of f and g to be '$Int \xrightarrow{\{F,G\}} Int$', which contains the identifiers for both of the lambdas.

The problem is that this step affects the type of f on the last line. The call of f on the final line gives us the set $\{F, G\}$, which indicates that both lambdas labeled F and G are evaluated. This is an overestimation, as the definition of f only contains the lambda F. We would like f to have the type '$Int \xrightarrow{\{F\}} Int$', but it does not due its usage in h. We say that the usage of f in h poisons the annotation of f. Subeffecting and subtyping aim to resolve this problem.

2.3.2 Subeffecting and Subtyping

With subeffecting we allow the enlarging of effects as seen in Equation 2.3, where ψ would be the effect of a subexpression (e.g. a branch of an if-then-else) and ψ' would be the effect derived for the encapsulating expression (e.g. the if-then-else). For our example in Section 2.3.1 this means that we can now unify the branches in h without poisoning f and g. In h we can give the if-then-else the type '$Int \xrightarrow{\{F,G\}} Int$', because $\{F\} \subseteq \{F, G\}$ and $\{G\} \subseteq \{F, G\}$, while f and g keep the types '$Int \xrightarrow{\{F\}} Int$' and '$Int \xrightarrow{\{G\}} Int$' respectively. The sets on the arrow of f and g are *subeffects* of the set on the arrow of the if-then-else type.

$$\frac{}{\psi \subseteq \psi'} \tag{2.3}$$

Subtyping, as the name suggests, allows weakening of the types. With subtyping we define a partial order over the annotated types. Note that the types in Equation 2.4 are annotated types. Subtyping can often be done at a later point than subeffecting. This makes subtyping stronger, but a bit more logic is required to implement it. Both subtyping and subeffecting can be generalized for elements ψ and ψ' of a lattice $\mathcal{L}$ with order $\sqsubseteq$ to $\psi \sqsubseteq \psi'$.

$$\frac{}{\hat{\tau} \leq \hat{\tau}} \qquad \frac{\hat{\tau}_1' \leq \hat{\tau}_1 \quad \hat{\tau}_2 \leq \hat{\tau}_2' \quad \psi \subseteq \psi'}{\hat{\tau}_1 \xrightarrow{\psi} \hat{\tau}_2 \leq \hat{\tau}_1' \xrightarrow{\psi'} \hat{\tau}_2'} \tag{2.4}$$

2.3.3 Polyvariance

Another technique for improving analysis accuracy is polyvariance. With polyvariance we allow polymorphism in the annotations. This means we extend our annotation language with annotation variables and quantification, as seen in Figure 2.5.

$$\begin{aligned} \beta &\in \textbf{AnnVar} \\ \phi, \psi &::= \beta \mid \{\pi\} \mid \ldots \\ \hat{\tau} &::= \forall \beta . \hat{\tau} \mid \mathit{int}\langle \psi \rangle \mid \ldots \end{aligned}$$

Figure 2.5: A polyvariant type and effect system

The set **AnnVar** contains all annotation variables and β is such a variable. Polyvariance especially increases accuracy for functions like '`id`'. In the CTA example we could then give a function '`idF::(a -> a) -> (a -> a)`' the annotated type in Equation 2.5. Applying `idF` to some function does not change the annotation of that function, which is reflected in the annotated type.

$$idF :: \forall \beta .\ (a \xrightarrow{\beta} a) \xrightarrow{\{IDF\}} (a \xrightarrow{\beta} a) \tag{2.5}$$

2.4 Going higher ranked

Higher ranked polymorphism allows polymorphism anywhere in a type, not just on the top level. If we only allow quantification at the top level in a type it is called rank-1 polymorphism. Type inference of Rank-N programs (with $N > 2$) is undecidable [7], so user annotations are required for type inference.

Consider the function `f` from Equation 2.6 for which the quantifications for `b` and `c` on the top level are implicit.

```
f :: (forall a .a -> a) -> b -> c -> (b, c)
f h x y = (h x, h y)
```
(2.6)

The quantification on the function parameter cannot be moved to the outer scope without changing the meaning of the type. If we would move the '`forall a`' outside of the function would not type check anymore, because we can instantiate the type variable `a` with Int and `b` with $Char$, which would make the application of x to h ill-typed. If we do not change the position of the '`forall a.`', then `h` is forced to be polymorphic in `a`: it must accept any type.

Higher ranked polyvariance allows type *annotations* to be higher ranked. Depending on the analysis a higher-ranked analysis can be decidable. Basically, this is because the types are already determined. That information can be used to supplement the higher-ranked analysis [8, 9, 10]. We will discuss a framework for higher-ranked analysis in Section 3.1.

3 Related work

3.1 Higher-ranked type & effect systems

In his master thesis, Thorand described a higher-ranked type and effect framework to perform dependency analysis [10]. Thorand introduces an *annotation language* $\lambda^{\sqcup}$ as seen in Figure 3.1. Note that the annotations become a lambda calculus in themselves. This includes rules for evaluating annotations, types and typing rules for the annotations, which are called sorts and sorting rules. The framework is defined over some lattice L.

$$\begin{array}{llr}
\ell \in \mathcal{L} & & \textit{(Lattice)} \\
\beta \in \mathbf{AnnVar} & & \textit{(Annotation variable)} \\
\xi \in \mathbf{AnnTm} & ::= \beta & \textit{(Variable)} \\
 & \mid \lambda\beta.\xi & \textit{(Abstraction)} \\
 & \mid \xi_1\ \xi_2 & \textit{(Application)} \\
 & \mid \xi_1 \sqcup \xi_2 & \textit{(Lattice join)} \\
 & \mid \ell & \textit{(Lattice value)}
\end{array}$$

Figure 3.1: The $\lambda^{\sqcup}$ calculus

This annotation language can be embedded in the terms by expanding the language with abstraction and application of annotations (Figure 3.2). This allows for applications in the terms to be reflected in the annotations. The annotated types are the same as the polyvariant type and effect system from Section 2.3.3, but without effects on the arrow. These are no longer required due to the embedding in the terms.

$$\begin{array}{lll}
\hat{t} \in \widehat{\mathbf{Tm}} & ::= \ \ldots & \\
 & \mid \Lambda\beta.\hat{t} & \textit{(Annotation abstraction)} \\
 & \mid \hat{t}\langle\xi\rangle & \textit{(Annotation application)}
\end{array}$$

Figure 3.2: Annotation abstraction and application in terms

If we take a look at the id function we can see how the type level polyvariance (not polymorphism) is reflected in the term by a lambda 'Λ' in the definition. The function `id` is not a higher-order function, so we do not yet gain extra precision. From the annotated type we can clearly see that the resulting x must have the same effect as the input.

When the polyvariant effect variable β is instantiated with some effect ψ, then β is instantiated in the type *and* in the expression. In the expression we see that parameter x gets annotated with the type *int* and effect ψ. The annotated type becomes '$int\langle\psi\rangle \to int\langle\psi\rangle$'. This x is returned by the lambda, with the same type and effect.

$$\begin{aligned}&\texttt{id :: } \forall\beta.\text{int}\langle\beta\rangle \to \text{int}\langle\beta\rangle\\ &\texttt{id = } \Lambda\beta.(\lambda x : \mathit{int} \ \&\ \beta.\ x)\end{aligned}$$

If we come back to the CTA example from Section 2.3 we can see why the effect on the arrow is no longer required. The effect of the body is now reflected by the terms. The effect set β is extended with the evaluated lambda ID.

$$\begin{aligned}&\texttt{id :: } \forall\beta.\text{int}\langle\beta\rangle \to \text{int}\langle\beta \sqcup \{ID\}\rangle\\ &\texttt{id = } \Lambda\beta.(\lambda_{ID} x : \mathit{int} \ \&\ \beta \sqcup \{ID\}.\ x)\end{aligned}$$

3.1.1 Higher ranked polyvariance

The type annotations for higher order functions get quite large even for rather simple functions. Take `($)` for instance as seen below. Note that even though `($)` is not higher ranked, the derived annotated type does contain higher ranked polyvariance. The parts referring to the higher order argument are written in grey.

The important concept to take away from this annotation is all the way at the end: $f\langle\beta_3\rangle\ x$. At this point we apply the inferred effects of x (β_3) to the polyvariant argument β of f.

$$\begin{aligned}&\texttt{(\$) :: } \forall\beta_1,\beta_2.(\forall\beta.\text{int}\langle\beta\rangle \to \text{int}\langle\beta_2\ \beta\rangle)\langle\beta_1\rangle) \to (\forall\beta_3.\text{int}\langle\beta_3\rangle \to \text{int}\langle\beta_2\ (\beta_3 \sqcup \beta_1)\rangle)\\ &\texttt{(\$) = } \Lambda\beta_1.\Lambda\beta_2.\lambda f : (\forall\beta.\text{int}\langle\beta\rangle \to \text{int}\langle\beta_2\ \beta\rangle).\ \Lambda\beta_3\lambda x : \beta_3.f\langle\beta_3\rangle\ x\end{aligned}$$

This localized application prevents poisoning in a function like '`both ::` $(a \to b) \to (a,a) \to (b,b)$' because the polyvariant argument is applied separately for each of the tuple elements. For a derivation of the annotations of `both` we suggest reading Section 5.3 of Thorands thesis [10].

3.1.2 Fixpoints

A problem with higher ranked analyses is deriving annotations for recursive functions. When a function is recursive its corresponding annotation becomes recursive as well. Thorands [10] and Wolffs [11] theses aim to solve this problem using fixpoint iteration.

To do this we must define a lattice over our annotation language. One can rewrite a recursive annotation as the fixpoint of a function $\mu : \beta \to \beta$. This function can be used for fixpoint iteration, starting with '$\mu\ \bot$', where '$\bot$' is the bottom on the annotation lattice.

The main problem is deciding *if* a fixpoint has been reached. To check if we have reached a fixpoint, one has to verify if two lambda terms are equal. This is difficult for higher order functions, because the annotation of such a function contains an abstraction.

Thorand solves this by checking all possible inputs for the annotation functions and comparing the output. This approach becomes rather expensive if the domain of inputs is large. The set of inputs is finite in Thorands thesis, because underlying language does not have polymorphism. However, if we introduce polymorphism the set of possible inputs becomes infinite.

De Wolffs thesis constructs an analysis on a calculus with polymorphism. To solve higher order fixpoints De Wolff delays solving of fixpoints until the higher order arguments of a function are supplied. This does not work in all cases, which means that widening might still be required.

3.2 Region Based Memory

The concept of region based memory management is quite old [12]. Initially region management had to be done by the programmer. Influential work was done on the ML Kit compiler for which Tofte and Birkedal created a region inference system [4]. The work done on ML Kit inspired many other implementations for memory-safe C dialects [13, 14, 15], logic languages like Prolog [16] and Mercury [17], and Real-Time Java [18].

3.2.1 Standard ML

The MLKit compiler [19] has an implementation with region based memory management that has been worked on extensively [4]. The implementation has automatic region inference and region size inference. The region inference automatically finds good positions to insert `letregion` expressions. The analysis also annotates expressions with $get(\rho)$ and $put(\rho)$ effects, these are the atomic effects. $get(\rho)$ denotes that there will be read from ρ and $put(\rho)$ denotes that some value will be put into ρ. A multiplicity analysis then uses those atomic effects to infer the size of those regions. Recently the implementation has been extended to be combined with a garbage collector [20]. Hallenberg implemented a profiler to get more insight into how regions behaved [21].

Multiplicity analysis

To infer the size of regions Vejlstrup designed and implemented a region size inference [22]. The region size inference takes in the annotated types from the region inference. Multiplicities are a finite set of numbers $\{0, 1, \ldots, K\} \cup \{\infty\}$ where K is a parameter of the analysis. Basically, the analysis infers the number of times values are written to each region. If the analysis finds a multiplicity larger than K, the multiplicity is set to ∞. This guarantees termination of the analysis, because the region sizes cannot grow infinitely (the lattice is finite, which implies the ascending chain condition). Vejlstrup concludes that $K = 1$ is sufficient as bounded regions with a size other than 0 or 1 are extremely rare. Veljstrup implemented a custom refinement method for effect sets, because the annotations are always evaluated completely for subexpressions, which causes loss of precision in some cases.

Performance

The RBMM implementation in the MLKit works well. It is capable of handling the vast majority of memory management in typical standard ML programs. Around 90% of regions had an inferred size of 1 for the inspected programs [22]. Unbounded regions are problematic as there is no natural size for a region page. If we pick a large page size, then memory is wasted for unbounded regions that only have a small number of values stored in them. This leaves most of the region page empty. The unused space is called *region waste* and it can be as high as 20% of the total memory usage [4]. Picking a small page size requires a lot of region page allocations for regions that grow (e.g. constructing a list).

Some caution is required when writing code for a compiler that uses region inference. The authors wrote a guidebook for programmers to understand how to write code in a region-friendly manner [23]. Regions that are not managed well by RBMM could be handled by a garbage collector [20].

3.2.2 Haskell

The current industry standard Haskell compiler, the Glasgow Haskell Compiler, uses generational garbage collection. The new backend [24] of the Helium compiler does not yet have a memory management system. However, recently work has started on a region based approach. So far a higher-ranked region inference analysis has been implemented by De Wolff [11]. This thesis introduces higher ranked region size inference.

The region inference is implemented in the new backend of Helium. This backend first desugares Haskell into Heliums Core language. The Core language is a simplified Haskell-like language. The Core code is optimized with a set of analyses, after which it is translated into Iridium. Iridium is a strict imperative language with explicit memory management. The region analysis is implemented on Iridium.

Haskell versus ML

ML and Haskell are quite similar. Bot languages for instance have a Hindley-Milner type system and support for higher order functions. There are also a couple of key differences between ML and Haskell.

Laziness: Haskell is lazy and ML is strict. This means that ML will always immediately evaluate an expression and Haskell will not until it has to know the result.

Referential transparency: Haskell has referential transparency while ML does not. Referential transparency means that for any assignment '`x = expr`' we can substitute `x` for `expr` (and vice versa) without changing the result of an expression.

Side effects: Haskell does not have side effects (outside the IO monad) while ML does. For any function in Haskell that does not use the IO monad you are guaranteed the same results for the same parameters.

3.2.3 General issues

In general region based memory management faces some issues. The first issue is the region waste of unbounded regions, for which there is still no good solution. The second issue has to do with tail recursion which we will discuss in Section 3.3

3.3 Region lifetime

Tail recursive functions can be optimized into loops, this is called tail call optimization. A tail recursive functions only calls itself in the return positions of said function (e.g. 3.1). Because the function is called in a tail position, all local variables fall out of scope, which means we can reuse this space by turning the function into loop. This reduces the size of the stack.

$$f\ x = \textbf{if}\ x > 0\ \textbf{then}\ f\ (x-1)\ \textbf{else}\ x \tag{3.1}$$

When we introduce regions into the language we run into a dilemma. Do we infer regions before or after tail call optimization? If we introduce regions *before* tail call optimization functions calls are no longer in the return positions. The function is no longer tail recursive and thus not tail call optimizable. If we infer regions *after* tail call optimization, the local variables are assigned in a loop, which means they are assigned to a possibly infinite number of times. This makes the regions unbounded.

If we could *reset* regions in loops, we can derive finite bounds on region variables in loops. This problem is more general than just tail call recursion. The ability to reset and reuse regions allows us to give regions a shorter lifetime, which gives better memory bounds. Some proposals have been made to solve this issue, we discuss these in the next section.

3.3.1 Region lifetime analyses

To allow resetting regions Tofte and Birkedal designed and implemented *storage mode analysis* in the ML Kit. The analysis replaces the `at`-expression with `attop` and `atbot` [25]. An `attop` acts the same as an `at`. An `atbot` resets a region and adds the value to the start (bottom) of a region. However, the authors admit that the analysis was very complex and vulnerable to small program changes [4].

An approach by Crary et al. uses a uniqueness analysis [26]. In this paper Crary defines rules over the operational semantics of a lambda-style calculus with low level memory constructs. A set of region variables C is maintained throughout these rules, this is called the set of *capabilities*. Note that these regions do not have a lexical scope. The expression '`newrgn` ρ, x' allocates a region with name ρ at memory location x and adds the region to C. If some region variable ρ is in C it is allocated and can be used. Region variables are removed from C with a '`freergn` v' where v points to the location of the region. Problems arise with a function definition like 3.2.

$$\forall[\rho_1, \rho_2](\texttt{int at } \rho_1, \texttt{int at } \rho_2)\ \{\ \ldots\ \} \tag{3.2}$$

If some variable y of type *int* allocated in a region called ρ_3 is passed as both arguments to the function it will be aliased by ρ_2 and ρ_1. Thus Crary assigns a multiplicity to each capability in the form of ρ_n^1 or ρ_n^+. ρ_n^+ means that region variable ρ_n has more than one alias (multiplicity '+'). Deallocation of ρ_n is only allowed if ρ_n^1 occurs in C, in other words, if ρ_n does not have another alias. Subeffecting is used to enlarge C in a function call. Uniqueness can be maintained in a function call as described in 3.2 by defining a constraint between the parameter region variables and the capabilities. Crary does not present any performance results.

Aiken et al. deals with the alias problem in a different way [27]. Each region is given a color. Any alias of such a region is given the same color. An execution order is decided upon and constraints are generated based on when regions are written to and read from. These constraints are solved such that regions can be allocated and deallocated at safe locations, while minimizing region lifetime. This constraint based approach beats storage modes in every test. An issue with the analysis is that a part of it runs in worst-case exponential time, however in practice the complexity seemed to be similar to the storage modes solution. The authors were unable to deduce whether that is the case in general. The global nature of the algorithm also makes separate compilation difficult.

An approach by Henglein et al. keeps count of the number of references to some region [28]. They embed an imperative sublanguage into a lambda-style calculus. The embedded language has the following four operations:

- $[\![\textbf{new } \rho]\!]$: Allocate a new region with reference count 1.
- $[\![\textbf{release } \rho]\!]$: Decrement the reference count of the region assigned to ρ by 1.
- $[\![\rho' := \textbf{alias } \rho]\!]$: Assign the region of ρ to ρ' and increase the reference count by 1.
- $[\![\rho' := \rho]\!]$: Equivalent to $[\![\rho' := \textbf{alias } \rho]\!]$ $[\![\textbf{release } \rho]\!]$.

If the number of references to any region drops to 0 it is deallocated. The operations of the embedded language can be put before or after any expression. A region inference system was constructed using backwards abstract interpretation

3.4 Static resource analysis

Another branch of research is aimed at the static determination of amortized resource bounds. It has been shown that one can deduce linear [29], polynomial [30] and multivariate bounds [31] from a program. The analyzed resource can for instance be memory usage or execution time. All papers take a similar approach. The goal is to put a bound on the *potential* of a program. The potential has to be non-negative during the entire execution. They collect a set of resource constraints with an annotated operational semantic, for example the rule for a constant integer value n in Equation 3.3 from the paper on linear bounds [29]. The important part is the annotations on the turnstile. The potential has to be at least 'KmkInt' before allocating an integer. A total of m' potential remains.

$$\frac{n \in \mathbb{Z} \quad \ell \notin dom(\mathcal{H})}{\mathcal{V}, \mathcal{H} \,\big|\!\frac{m' + \text{KmkInt}}{m'}\, n \rightsquigarrow \ell, \mathcal{H}[\ell \mapsto (\text{int}, n)]} \,\texttt{(OP Const Int)} \tag{3.3}$$

The constraints will make sure that the potential at the start of the program is the total potential of the program. In other words, the potential at the start of the program is equal to the total cost of executing the program. The analyses are generic in the sense that different types of resource bounds can be inferred by altering the constant cost for certain operations. For instance, the constant 'KmkInt', the cost for making an integer, would be 1 in the calculation for stack space, but 0 for heap space. In this manner bounds were presented on the number of procedure calls, stack space, heap space and time.

The estimated bounds for memory are good, however sometimes linear bounds were estimated while constant bounds were measured. Time bounds were much less consistent than the other analyses. Time bounds were overestimated by up to 30%.

Static resource analyses is most useful for unbounded regions in RBMM. We could alter the behavior of heap allocation based on the computed bounds. If a bound is linear we could for instance allocate smaller region pages than in the case of a polynomial bound.

4 Research questions

We formulate the following research questions.

Question 4.1. *Can region bound inference be made higher ranked?*

We must verify if we can make the region bound inference higher ranked or if the problem becomes undecidable. We must also verify if going higher ranked yields more precision in a real world language such as Haskell.

Question 4.2. *Does RBMM have a significant impact on compilation speed?*

What is the overhead that RBMM creates when compiling a program? We can compare the speed of compiling programs without memory management, with region inference and with region inference *and* region bound inference.

Question 4.3. *Does RBMM have a significant impact on program speed?*

What is the overhead that RBMM creates when running a program? We can compare the speed of Helium-compiled programs without memory management, with region inference and with region inference *and* region bound inference.

5

Source language

We define a lambda calculus as the source language for our analysis. We make use of System F_ω, which is System F [32] extended with datatypes. System F has explicit quantification and instantiation of types in the terms, which is why they are also part of this calculus.

The calculus includes the **letregion** and **at** constructs. A **letregion** defines a region and an **at** puts a value in that region. The positions for these constructs have been inferred by the analysis of De Wolff [11].

An abstraction brings regions $\hat{\rho}$ and a variable x of type τ in scope for use within the abstraction. These region variables are called the *return regions* for that lambda. A recursive abstraction does the same, but f is in scope in the body. An application must also pass the region variables $\hat{\rho}$.

Datatypes can be constructed with constructors and destructed by a **case**. If the value of x of type D is matched to the i-th constructor of D, then the fields of said value are bound to a list of variables $\bar{x}_i$ which are in scope in the corresponding expression e_i. We assume there is a match for all constructors of the provided datatype.

$$\begin{array}{lllr}
e & ::= & x & \textit{(variable)} \\
 & \mid & n & \textit{(integer)} \\
 & \mid & \lambda[\hat{\rho}]x : \tau.\ e & \textit{(abstraction)} \\
 & \mid & \mu f : \tau\ .\lambda[\hat{\rho}]x : \tau.\ e & \textit{(recursive abstraction)} \\
 & \mid & e_1[\hat{\rho}]\ e_2 & \textit{(application)} \\
 & \mid & \forall\alpha.e & \textit{(quantification)} \\
 & \mid & e\ \{\tau\} & \textit{(type application)} \\
 & \mid & \textbf{if}\ c\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \textit{(if-then-else)} \\
 & \mid & \textbf{let}\ x = e\ \textbf{in}\ e & \textit{(let)} \\
 & \mid & (\ ,\ ) \mid (\ ,\ ,\ ) \mid \ \ldots & \textit{(tupling with arity N >1)} \\
 & \mid & \pi_i(e) & \textit{(projection)} \\
 & \mid & \textbf{letregion}\ \rho\ \textbf{in}\ e & \textit{(let-region)} \\
 & \mid & e\ \textbf{at}\ \rho & \textit{(at)} \\
 & \mid & D & \textit{(constructor)} \\
 & \mid & \textbf{case}\ x : D\ \textbf{of}\ D_1\ \bar{y}_1 \rightarrow e_1; \ldots ; D_n\ \bar{y}_n \rightarrow e_n & \textit{(case)}
\end{array} \tag{5.1}$$

The let, if-then-else, quantification, instantiation, tupling and projection constructs all behave as one would expect. The calculus uses lazy semantics.

5.1 Types

We will now introduce the type system of our calculus. The type system has N-tuples and polymorphism through type variables and quantification. There are strict types ($!\tau$) which indicate that the computation of a value will be strict. Datatypes may be polymorphic (e.g. lists). The polymorphic arguments of datatypes are instantiated by a list of types. The language supports higher order polymorphism.

$$\begin{aligned}
\tau \quad ::= \ & Int && \textit{(integer)} \\
\mid \ & (\rightarrow) && \textit{(function type)} \\
\mid \ & !\tau && \textit{(strict type)} \\
\mid \ & \tau\ \tau && \textit{(type application)} \\
\mid \ & (\,,\,) \mid (\,,,\,) \mid \ldots && \textit{(tuple of arity N >1)} \\
\mid \ & () && \textit{(unit, tuple of arity 0)} \\
\mid \ & \alpha && \textit{(type variable)} \\
\mid \ & \forall\alpha.\ \tau && \textit{(quantified type)} \\
\mid \ & D && \textit{(constructor)}
\end{aligned} \tag{5.2}$$

5.2 Datatypes

Datatypes are defined as a sum-of-products. They are defined separately from expressions and types. Datatypes can have type arguments and they can be (mutually) recursive. We leave out the syntax of datatype declarations, as they are equivalent to Haskell's algebraic datatypes. A simple example of a datatype definition can be found in Figure 5.1.

Figure 5.1: Example datatype definitions

```
data MaybeInt = Nothing
              | Just Int

data List a = Nil
            | Cons a (List a)
```

6 Region variables

The goal of the region bound analysis is to produce a mapping from each region to its size. In this chapter we define this mapping. We also define operators to manipulate such mappings and a partial order relation and a lattice.

$$\begin{aligned} \hat{\rho} \quad &::= \rho && \textit{(region)} \\ &\mid (\hat{\rho}, \hat{\rho}, \dots) && \textit{(tuple)} \end{aligned} \tag{6.1}$$

6.1 Regions

In a method annotation there are two kinds of regions: local regions and region variables. Local regions only exist in the method body; they are allocated and deallocated in the same scope. Region variables are bound by an abstraction and must be passed to the method; their lifetime outlives the method scope.

There are two special regions: ρ_{global} and ρ_{bottom}. The region ρ_{global} is the region that exists for the entire duration of the program. The region ρ_{bottom} is never allocated and can be used to discard or ignore allocations.

6.2 Constraint set

A constraint set is a mapping from regions to bounds. The notation for constraint sets can be found in Equation 6.2. The letter ρ denotes some region and $n \in \{0, 1, 2, 3..., K\} \cup \{\infty\}$ the bound. K is an analysis parameter, denoting the maximum size of a region before it is set to ∞, where ∞ denotes that no bound could be derived on said region. All regions are implicitly mapped to zero.

$$\{\ \rho^* \mapsto n\ \} \tag{6.2}$$

Because regions could be supplied as a tree of tuples we need to introduce projection to allow for a bound on specific sub trees or regions. The notation for projection is presented in Equation 6.3. Note that ρ in the example must be a region variable, as we cannot project on a local region.

$$\rho^* := \rho^*.i \mid \rho \tag{6.3}$$

6.2.1 Constraint set operators

The bound of some region ρ can be retrieved by applying the region to the constraint set. From now on we will call this 'indexing the constraint set' and we will use the notation '$\phi[\rho]$' to indicate that we index the constraint set ϕ with the region ρ. If $\rho \mapsto n \in \phi$ the indexing operation $\phi[\rho]$ will return n. If the mapping ϕ does not contain an explicit bound for ρ, indexing the mapping will return the implicit bound of zero.

The set of all possible mappings over some domain A will be denoted as L_A. The domain A will be used to represent all regions in scope.

The $\oplus$-operator is defined as the addition of two constraint sets. The operator adds up the bounds of identical regions. Formally, for any two constraint sets $\phi, \psi \in L_A$ the addition is defined as seen in Equation 6.4.

$$\phi \oplus_A \psi = \{\ \rho \mapsto \phi[\rho] + \psi[\rho] \mid \rho \in A\ \} \tag{6.4}$$

The $\backslash$-operator is defined to remove the bound on a region ρ from a mapping ψ. This operation is used to reset the region bound in case such a region is in a loop. Formally, for any constraint set $\phi \in L_A$ the restrict operation is defined as seen in Equation 6.5.

$$C \setminus_A \rho_i = \{\ \rho \mapsto C[\rho] \mid \rho \neq \rho_i, \rho \in A\} \tag{6.5}$$

6.3 Lattice

As before L_A denotes the set of all possible constraint sets over some domain A. We will now define the partial order relation $\preceq_A$.

Definition 6.3.1. *For any two constraint sets $\phi, \psi \in L_A$ the relation $\phi \preceq_A \psi$ holds iff for all $\rho \in A$ it holds that $\phi[\rho] \leq \psi[\rho]$.*

We say ϕ and ψ are incomparable if neither $\phi \preceq_A \psi$ or $\psi \preceq_A \phi$ holds. With this definition we can define the lattice over constraint sets $\mathcal{L}_A = (L_A, \preceq_A)$.

The '$\sqcup_A$'-operator is the join operator and is defined over any two constraint sets $\phi, \psi \in L_A$. The join takes the highest bound of identical regions. Intuitively the join can be seen as the maximum of two constraint sets. The formal definition is presented in Equation 6.6.

$$\phi \sqcup_A \psi = \{\ \rho \mapsto max(\phi[\rho], \psi[\rho]) \mid \rho \in A\ \} \tag{6.6}$$

6.3.1 Top & bottom

Bottom is equivalent to the empty constraint set as the lower bound of zero is implicit. Because of this equivalence it is not required to define bottom over the domain A as presented in Equation 6.7. Top, however, does require the domain and maps all regions in A to an unbounded size as presented in Equation 6.8

$$\bot_A = \{\rho \mapsto 0 \mid \rho \in A\ \}\ = \{\} = \bot \tag{6.7}$$

$$\top_A = \{\rho \mapsto \infty \mid \rho \in A\ \} \tag{6.8}$$

Using these definitions we can prove the following lemmas.

Lemma 6.3.1. *Bottom is the identity of join:* $\psi \sqcup \bot = \psi$.

Proof. For bottom $\bot[\rho] = 0$ for all ρ holds, because bottom does not put any bound on any region. For every region ρ we have that '$\psi[\rho] = n$' where $n \in \mathbb{N} \cup \{\infty\}$, which implies '$max(n, 0) = n$'. Thus for any constraint set ψ, $\bot$ will not make the constraints any stricter, which implies $\psi \sqcup \bot = \psi$. $\square$

Lemma 6.3.2. *For any set* $\psi \in L_A$ *it holds that* $\psi \sqcup \top_A = \top_A$.

Proof. The proof is dual to the proof of bottom. No bound is greater than ∞, thus for any bound that ψ puts on regions from the domain A, the bound of $\top_A$ will be greater. $\square$

6.4 Domain and scope

As mentioned before the domain A is used to represent all regions in scope. This set changes dynamically throughout the analysis of a program as regions are brought in and out of scope. For ease of readability we will omit the subscript A from the addition, restrict, partial order and join operators. In any case where the domain A is omitted one can assume that it is equal to all regions in scope.

7 Annotation sorts

Before we present our annotation language (Chapter 8), we will first present the type system for the annotation language. For this purpose we follow the terminology used by Thorand [10] and De Wolff [11] and we will call the types of our annotation language *sorts*.

There is a function, quantification, unit and tuple sort as there are with the type system of the source language. Note that quantifications provide a *type* variable α, not a sort variable. These type variables are used in polymorphic regions ($P\langle\alpha\ \bar{\tau}\rangle$) and polymorphic sorts ($\Psi\langle\alpha\ \bar{\tau}\rangle$) which we will elaborate on in Section 7.1. Constraint sets are of sort C and regions of sort P.

$$\begin{aligned}
s \quad ::=\ & s \mapsto s && \textit{(function)} \\
\mid\ & () && \textit{(unit)} \\
\mid\ & (s,s) \mid (s,s,s) \mid \ \ldots && \textit{(tuple)} \\
\mid\ & C && \textit{(constraint set)} \\
\mid\ & P && \textit{(region)} \\
\mid\ & \forall\alpha.\ s && \textit{(quantification)} \\
\mid\ & P\langle\alpha\ \bar{\tau}\rangle && \textit{(polymorphic region sort)} \\
\mid\ & \Psi\langle\alpha\ \bar{\tau}\rangle && \textit{(polymorphic sort)}
\end{aligned} \tag{7.1}$$

7.1 Sort assignment

As discussed in the introduction, higher ranked type inference is undecidable. The terms of the source language are annotated with types, which make it possible to derive a higher ranked type. Deriving higher ranked sorts from annotations is undecidable as well, so we must convert the type annotations from the source language into sort annotations in the annotated language. For this purpose we define the region and sort assignment functions.

7.1.1 Region assignment

We define the function $P_\Gamma : \tau \to s$ to assign regions to a type. The sort s defines the regions required to store some data with type τ. The function is also supplied with a datatype environment Γ, which contains the sorts for the datatypes. How Γ is created will be discussed in Section 7.3.

A type requires two regions: a place to store the thunk and a place to store the computed value. Strict types ($!\tau$) do not create a thunk, which means we can omit that region. Some types require *nested regions*, which are assigned by the function $P^\circ_\Gamma : \tau \to s$.

$$\begin{aligned} P_\Gamma(!\tau) &= (P, P^\circ_\Gamma(\tau)) \\ P_\Gamma(\tau) &= (P, P, P^\circ_\Gamma(\tau)) \end{aligned} \tag{7.2}$$

Whenever we want to store a value in multiple parts we use the nested regions. Integers and units exist out of only one part, which means no nested regions are required. An abstraction does not store any data, storing the result is discussed in Section 7.1.2. Tuples require nested regions for the thunks and values of the nested types. The region assignment is equivalent to the one presented by De Wolff [11].

$$\begin{aligned} P^\circ_\Gamma(Int) &= () \\ P^\circ_\Gamma(()) &= () \\ P^\circ_\Gamma(\tau_1 \to \tau_2) &= () \\ P^\circ_\Gamma((\tau_0, \ldots, \tau_n)) &= (P, P, P^\circ_\Gamma(\tau_0), \ldots, P, P, P^\circ_\Gamma(\tau_n)) \\ P^\circ_\Gamma(\alpha\ \tau_1 \ldots \tau_n) &= P\langle\ \alpha\ \tau_1 \ldots \tau_n\ \rangle \\ P^\circ_\Gamma(D\ \bar{\tau}) &= \text{The regions assigned to datatype } D\ \bar{\tau} \text{ in } \Gamma \end{aligned} \tag{7.3}$$

7.1.2 Sort assignment

We define a function $\Phi_\Gamma : \tau \to s$ to assign a sort to a type. Abstractions are special, as they also need to store the effect of the computation. The effect of a computation is represented as a constraint set C. We abstract over this effect with a region variable of sort $P_\Gamma(\tau_2)$. We refer to this region as the return region of a function.

$$\Phi_\Gamma(\tau_1 \to \tau_2) = \Phi_\Gamma(\tau_1) \mapsto P_\Gamma(\tau_2) \mapsto (\Phi_\Gamma(\tau_2), C) \tag{7.4}$$

The other sort assignments are trivial. Base types get the unit sort, tuples stay tuples and polymorphic types get polymorphic sorts. A strict type has the same sort as a non-strict type.

$$\begin{aligned} \Phi_\Gamma(Int) &= () \\ \Phi_\Gamma(()) &= () \\ \Phi_\Gamma((\tau_0, \ldots, \tau_n)) &= (\Phi(\tau_0), \ldots, \Phi(\tau_n)) \\ \Phi_\Gamma(!\tau) &= \Phi(\tau) \\ \Phi_\Gamma(\alpha\ \tau_1 \ldots \tau_n) &= \Phi\langle\ \alpha\ \tau_1 \ldots \tau_n\ \rangle \\ \Phi_\Gamma(\forall \alpha.\tau) &= \forall \alpha.\Phi(\tau) \\ \Phi_\Gamma(D\ \bar{\tau}) &= \text{The sort assigned to datatype } D\ \bar{\tau} \text{ in } \Gamma \end{aligned} \tag{7.5}$$

7.2 Type instantiation

Type instantiation is computed by a syntactic substitution presented in Equation 7.6. The rules for polymorphic (region) sorts instantiate the type variable if it matches the one of the substitution, after which Φ_Γ and P_Γ are used to further assign the sort.

$$\begin{aligned}
C[\alpha := \tau] &= C \\
P[\alpha := \tau] &= P \\
()[\alpha := \tau] &= () \\
(s_1 \mapsto s_2)[\alpha := \tau] &= s_1[\alpha := \tau] \mapsto s_2[\alpha := \tau] \\
(s_1, \dots, s_n)[\alpha := \tau] &= (s_1[\alpha := \tau], \dots, s_n[\alpha := \tau]) \\
(\forall \alpha'.s)[\alpha := \tau] &= \forall a.(s[\alpha := \tau]) \\
\Psi\langle \alpha' \; \tau_1, \dots, \tau_n\rangle[\alpha := \tau] &= \begin{cases} \alpha = \alpha' & \Phi_\Gamma(\tau \; \tau_1[\alpha := \tau], \dots, \tau_n[\alpha := \tau]) \\ \alpha \neq \alpha' & \Psi\langle \alpha' \; \tau_1[\alpha := \tau], \dots, \tau_n[\alpha := \tau]\rangle \end{cases} \\
\hat{P}\langle \alpha' \; \tau_1, \dots, \tau_n\rangle[\alpha := \tau] &= \begin{cases} \alpha = \alpha' & P_\Gamma(\tau \; \tau_1[\alpha := \tau], \dots, \tau_n[\alpha := \tau]) \\ \alpha \neq \alpha' & \hat{P}\langle \alpha' \; \tau_1[\alpha := \tau], \dots, \tau_n[\alpha := \tau]\rangle \end{cases}
\end{aligned} \tag{7.6}$$

7.3 Datatypes

When analyzing datatypes we devise three categories: simple datatypes, simple recursive datatypes and complex datatypes (e.g. the finger tree datatype by Hinze and Paterson [33]). Type class dictionaries are desugared into datatypes and analyzed with the same rules as datatypes.

7.3.1 Simple datatypes

Simple datatypes are non recursive and only exist out of product and sum types. When assigning a sort to a simple datatype a tuple is created with an annotation for each *field* (not constructor) in the datatype. For the simple datatype seen in Equation 7.7 a tuple would be created with region and sort assignments as seen in Equation 7.8.

$$\begin{aligned}
\textbf{data}\ \mathit{Foo}\ a\ b\ c = {} & \mathit{Bar}\ a\ b \\
\mid {} & \mathit{Baz}\ c \\
\mid {} & \mathit{Quz}\ b
\end{aligned} \tag{7.7}$$

$$\begin{aligned}
\Phi_\Gamma(\mathit{Foo}\ a\ b\ c) &= \forall a, b, c.\Phi_\Gamma(((a, b), (c), (b))) \\
P^\circ_\Gamma(\mathit{Foo}\ a\ b\ c) &= \forall a, b, c.P_\Gamma(((a, b), (c), (b)))
\end{aligned} \tag{7.8}$$

We define Equation 7.9 for assigning sorts to non-recursive datatypes and Equation 7.10 for assigning regions to non-recursive datatypes.

$$\Phi_\Gamma(\textbf{data}\ D\ \alpha_1 \dots \alpha_n = D_1\ \bar{\tau}_1 \mid \dots \mid D_n\ \bar{\tau}_n) = \forall \alpha_1. \dots \forall \alpha_n.\Phi_\Gamma((\bar{\tau}_1, \dots, \bar{\tau}_n)) \tag{7.9}$$

$$P^\circ_\Gamma(\textbf{data}\ D\ \alpha_1 \dots \alpha_n = D_1\ \bar{\tau}_1 \mid \dots \mid D_n\ \bar{\tau}_n) = \forall \alpha_1. \dots \forall \alpha_n.P_\Gamma((\bar{\tau}_1, \dots, \bar{\tau}_n)) \tag{7.10}$$

7.3.2 Simple recursive datatypes

Simple recursive datatypes are non-mutually recursive and assign their recursive polymorphic arguments the same way. Recursive datatypes pose a problem for sort assignment, as the sort of the datatype would be required to instantiate the sort of said datatype. To solve this the unit sort is passed as the sort of the datatype during sort assignment. These recursive positions also get a unit annotation.

$$P^{\circ}_{\Gamma,D:()}(\mathbf{data}\ D\ \alpha_1 \ldots \alpha_n = t) \tag{7.11}$$

The sort assignment for a list datatype `data List a = Cons a (List a) | Nil` is analyzed to the sort $\forall\alpha.(\Phi_\Gamma\langle\alpha\rangle, ())$. Note that simple datatypes do not include datatypes that are recursive through the contravariant position of functions.

7.3.3 Complex datatypes

We define complex datatypes as the class of datatypes that does not match the other two categories. This includes datatypes with mutual recursion, datatypes that assign their recursive argument with a different type and datatypes recursive in the contravariant position of a function.

Due to time constraints we have decided that complex recursive datatypes are out of scope for this thesis. This decision is motivated by the fact that analyzing such complex structures is unlikely to give an interesting result. The region analysis currently does not support these datatypes either.

We assign the unit sort to these datatypes. Whenever we construct such a datatype it will have a unit sort. When we destruct the datatype we will create a top annotation for each of the fields. Such a datatype will also be assigned one nested region to store any results.

8 Annotation language

The annotation language is presented in Equation 8.1. Just like the source language, the annotation language is a lambda calculus too. We will now go through all the elements of the language. Note that the language is essentially an extended version of the annotation language presented in Section 3.1.

An abstraction brings a variable ψ of sort s in scope. A variable ψ may be an annotation or a region variable. Local regions are represented by ρ. Note that constraint sets can now map variables to bounds.

The restrict, add and join operator are defined over annotations. Between two constraint set annotation they behave the same as the constraint set operators presented in Section 6.2.1.

Type quantification quantifies over *type* variables in polymorphic *sorts*. Instantiation substitutes a type variable α with the type τ. Top is annotated with constraint set C and a sort s. Bottom is only annotated with a sort s, because it does not put constraints on regions. The fixpoint is annotated with a list of sorts and a list of annotations.

$$
\begin{aligned}
a \quad ::= \;& \psi && \textit{(variable)} \\
\mid\; & \rho && \textit{(local region)} \\
\mid\; & \{\psi \mapsto n\} && \textit{(constraint set)}0 \\
\mid\; & \Lambda\psi : s.\ a && \textit{(abstraction)} \\
\mid\; & a\langle a\rangle && \textit{(application)} \\
\mid\; & \forall\alpha.\ a && \textit{(type quantification)} \\
\mid\; & a\ \{\tau\} && \textit{(type instantiation)} \\
\mid\; & () && \textit{(unit)} \\
\mid\; & (,) \mid (,,) \mid \ \ldots && \textit{(tupling)} \\
\mid\; & \pi_i[a] && \textit{(projection)} \\
\mid\; & a \oplus a && \textit{(add)} \\
\mid\; & a \setminus \rho && \textit{(restrict)} \\
\mid\; & \top[C : s] && \textit{(top)} \\
\mid\; & \bot[s] && \textit{(bottom)} \\
\mid\; & a \sqcup a && \textit{(join)} \\
\mid\; & fix : s.\ a && \textit{(fixpoint)}
\end{aligned}
\tag{8.1}
$$

8.1 Sorting rules

Just as there are typing rules for the source language, there are *sorting rules* for the annotation language. As the types of our annotation language are called sorts, typing rules for the annotation language are called the *sorting rules*. The notation '$\Gamma \vdash_s a : s$' is used to denote that the annotation a is *well sorted* under sort environment Γ.

8.1.1 Sort environment

We define a sort environment Γ to keep track of the sort of variables in scope. The sort environment can extended with a variable ψ with sort s as presented in Equation 8.2. The initial environment is empty: $\Gamma = \{\}$.

$$\Gamma := \{\} \mid \Gamma, \psi : s \tag{8.2}$$

8.1.2 Variables

The sort of a variable is stored in the sort environment Γ.

$$\frac{}{\Gamma, \psi : s \vdash \psi : s} \tag{8.3}$$

8.1.3 Abstraction and application

In an abstraction (8.4) the environment is extended sort of the variable that is brought into scope. The sort of the abstraction body is derived with this extended environment. The final sort is a sort lambda from the argument sort to the result sort.

$$\frac{\Gamma, \psi : s_a \vdash_s a : s_r}{\Gamma \vdash_s \Lambda\psi.\ a : s_a \mapsto s_r} \tag{8.4}$$

In an application (8.5) the result sort is returned, given that the argument sort of the argument x is equal to the expected argument sort. Note that we allow the application of a monomorphic region variable to a polymorphic region variable sort, but not the other way around.

$$\frac{\Gamma \vdash_s f : s_a \mapsto s_r \qquad \Gamma \vdash_s x : s_a}{\Gamma \vdash_s f\langle x\rangle : s_r} \tag{8.5}$$

8.1.4 Quantification and instantiation

A quantification is reflected in the sort α (8.6). Instantiations (8.7) are sorted by performing the substitution on the sort. Note that the sort of the sub-annotation must start with a quantification. This may require to do sort/region assignment on the instantiated type(s).

$$\frac{\Gamma \vdash_s a : s}{\Gamma \vdash_s \forall\alpha.a : \forall\alpha.s} \tag{8.6}$$

$$\frac{\Gamma \vdash_s a : \forall\alpha.s}{\Gamma \vdash_s a\ \{\tau\} : s[\alpha := \tau]} \tag{8.7}$$

8.1.5 Tuples

The sorting rules for tuples are straightforward. For a tuple of n elements (8.8) each element is sorted separately. The sort of the tuple is a sort tuple of the resulting sorts. With projection (8.9) the k-th sort is taken from the n-tuple, given that $k < n$. An annotation unit '()' is a zero-tuple, thus the sort of a unit is also a unit.

$$\frac{\Gamma \vdash_s a_0 : s_0,\ \ldots\ ,\Gamma \vdash_s a_{n-1} : s_{n-1}}{\Gamma \vdash_s (a_0, \ldots, a_{n-1}) : (s_0, \ldots, s_{n-1})} \tag{8.8}$$

$$\frac{\Gamma \vdash_s a : (s_0, \ldots, s_{n-1}) \qquad 0 \leq k < n}{\Gamma \vdash_s \pi_k(a) : s_k} \tag{8.9}$$

8.1.6 Join, top & bottom

An annotation is only well sorted if the join operands are of the same sort. The sort of the join is the same sort as its operands. The top and bottom annotation are annotated with their sort.

$$\frac{\Gamma \vdash_s a_1 : s \qquad \Gamma \vdash_s a_2 : s}{\Gamma \vdash_s a_1 \sqcup a_2 : s} \tag{8.10}$$

$$\frac{}{\Gamma \vdash_s \bot[s] : s} \tag{8.11}$$

$$\frac{}{\Gamma \vdash_s \top[C : s] : s} \tag{8.12}$$

8.1.7 Fixpoint

The sort of a fixpoint can be derived by constructing a lambda with a annotation variable with the sort s and the body a. A fixpoint is well sorted if and only if the sort of the argument is equal to the sort of the body. Not that the fixpoint must have a sort from s to s.

$$\frac{\Gamma \vdash_s \lambda\phi : s.\ a : s \mapsto s}{\Gamma \vdash_s (fix : s.\ a) : s \mapsto s} \tag{8.13}$$

8.1.8 Add

Both operands of the operator must be of the constraint set sort C, because addition is only defined over constraint sets. The result of the addition operator is also of sort C.

$$\frac{\Gamma \vdash_s a_1 : C \qquad \Gamma \vdash_s a_2 : C}{\Gamma \vdash_s a_1 \oplus a_2 : C} \tag{8.14}$$

8.2 Lattice

In Section 6.3 we defined a lattice over constraint sets. We will now define a lattice over annotations $\mathcal{A}_s = (A_s, \sqsubseteq_s)$. The set A_s is the set of all annotations of sort s and '$\sqsubseteq_s$' a partial order between its elements. Two annotations of different sorts are incomparable, as they are not even on the same lattice.

8.2.1 Top & bottom

The annotation language provides a top and bottom annotation. In fixpoint iteration the recursive argument is initialized with bottom. Fixpoint iteration can take very long and will never terminate in some cases. In either case we can decide to give up and default to top. Top is also used for destructing datatypes that we are unable to analyze.

Bottom is annotated with a sort, such that the sorting rules can still derive a sort for the annotation. Bottom is the least annotation in the lattice, such that for any annotation $\Gamma \vdash_s a : s$ it follows that $\bot[s] \sqsubseteq a$.

Top is annotated with a sort and a constraint set. The sort is required for the sorting rules. The constraint set contains the effect of top, which maps all local variables and region variables in scope to ∞. Top is the greatest annotation in the lattice, such that for any annotation $\Gamma \vdash_s a : s$ it follows that $a \sqsubseteq \top[C : s]$, where C must contain at least all variables and regions from a mapped to unbounded.

8.2.2 Partial order

We will now extend the partial order over the entire annotation language. For any two annotation terms a_1 and a_2 of equal sort s we say that a_1 is at least as precise as a_2 ($a_1 \sqsubseteq_s a_2$) if and only if one of the following conditions holds:

Unit $s = ()$: The only annotation with sort '$()$' is unit. So $a_1 = a_2 = ()$ and thus $a_1 \sqsubseteq_s a_2$ holds.

Constraint set $s = C$: We have $a_1 \sqsubseteq_s a_2$ if and only if it holds on the constraint set lattice: $a_1 \preceq a_2$.

Tuple $s = (s_0, \ldots, s_n)$: We have a $a_1 \sqsubseteq_s a_2$ if and only if for all indexes $0 \leq k \leq n$ we have $\pi_k(a_1) \sqsubseteq_{s_k} \pi_k(a_2)$.

Quantification $s = \forall\alpha.s_1$: We have $a_1 \sqsubseteq_s a_2$ if and only if for all types τ we have $a_1\{\tau\} \sqsubseteq_{s\{\tau\}} a_2\{\tau\}$.

Lambda $s = s_1 \mapsto s_2$: We have $a_1 \sqsubseteq_s a_2$ if and only if for all annotations a_3 of sort s_1 we have $a_1\langle a_3\rangle \sqsubseteq_{s_2} a_2\langle a_3\rangle$.

Lemma 8.2.1. *The set A_s is infinite.*

Proof. For all sorts s we have at least one annotation: $\bot[s]$. For any annotation $\Gamma \vdash_s a : s$, we can generate an infinite number of annotations of equal sort by simply wrapping the annotation with a lambda and an application, for example '$(\psi : ().a)\langle()\rangle$'. This maintains the sort and we can do this an infinite number of times. Thus any set A_s has an infinite number of elements. □

8.3 Evaluation rules

For the evaluation of our annotations a relation '$\longrightarrow$' is defined that is reflexive (8.15) and transitive (8.16). The notation $a_1 \longrightarrow a_2$ indicates that a well sorted annotation a_1 evaluates to a_2.

$$\frac{}{a \longrightarrow a} \tag{8.15}$$

$$\frac{a_1 \longrightarrow a_2 \qquad a_2 \longrightarrow a_3}{a_1 \longrightarrow a_3} \tag{8.16}$$

8.3.1 Sub-evaluations

Sub-annotations may have to be evaluated before the encapsulating annotations can be evaluated. Another reason to do sub-evaluations first is performance related, i.e. evaluation the argument of an application prevents having to evaluate it twice in case it is duplicated.

$$\frac{a \longrightarrow a'}{\Lambda\psi.a \longrightarrow \Lambda\psi.a'} \tag{8.17}$$

$$\frac{a_1 \longrightarrow a_1' \qquad a_2 \longrightarrow a_2'}{a_1\langle a_2\rangle \longrightarrow a_1'\langle a_2'\rangle} \tag{8.18}$$

$$\frac{a_1 \longrightarrow a_1' \qquad a_2 \longrightarrow a_2'}{(a_1, a_2) \longrightarrow (a_1', a_2')} \tag{8.19}$$

$$\frac{a \longrightarrow a'}{\pi_i(a) \longrightarrow \pi_i(a')} \tag{8.20}$$

$$\frac{a \longrightarrow a'}{\forall\alpha.a \longrightarrow \forall\alpha.a'} \tag{8.21}$$

$$\frac{a \longrightarrow a'}{a\ \{\tau\} \longrightarrow a'\ \{\tau\}} \tag{8.22}$$

8.3.2 Application

Applications as performed by a syntactic substitution on the annotation. The definition of the entire syntactic substitution can be found in Appendix A.

$$\frac{}{(\Lambda\psi.a_1)\langle a_2\rangle \longrightarrow a_1[\psi := a_2]} \tag{8.23}$$

Extra caution must be taken when a variable is substituted in a constraint set. Constraint set substitutions are not always a simple syntactic substitution. Each substitution is handled separately and added back to the constraint set as presented in Equations 8.24 and 8.24. We discern two cases: the annotation is a tree of regions or it is some other annotation.

$$\frac{}{(C \cup \{\psi \mapsto n\})[\psi := \rho] \longrightarrow C[\psi := \rho] \oplus \{\psi \mapsto n\}[\psi := a]} \tag{8.24}$$

$$\frac{\psi \notin C}{C[\psi := a] \longrightarrow C} \tag{8.25}$$

The annotation is a tree of regions

The simplest case is where a single variable is substituted for a single region as seen in Equation 8.26.

$$\frac{}{\{\psi \mapsto n\}[\psi := \langle\rho\rangle] \longrightarrow \{\rho \mapsto n\}} \tag{8.26}$$

Projection is handled by projecting on the tuple. Note that a single variable may project on a tree more than once (e.g. $\{\psi.1.2.1 \mapsto n\}$). Also note that each element of the tuple $(\hat{\rho_1}, \ldots, \hat{\rho_n})$ may be a tree of regions in itself as well.

$$\frac{1 \leq k < n}{\{\psi.k \mapsto n\}[\psi := (\hat{\rho_1}, \ldots, \hat{\rho_n})] \longrightarrow \{\psi \mapsto n\}[\psi := \hat{\rho_k}]} \tag{8.27}$$

It may occur that a variable is substituted with a tree that does not project on said tree. By using the function *collect* the tuple is converted into a constraint set with the same bound n as the variable is mapped to.

$$\frac{}{\{\psi \mapsto n\}[\psi := (\hat{\rho_1}, \ldots, \hat{\rho_n})] \longrightarrow \mathit{collect}((\hat{\rho_1}, \ldots, \hat{\rho_n}), n)} \tag{8.28}$$

$$\begin{aligned} \mathit{collect}(\rho, n) &= \{\rho \mapsto n\} \\ \mathit{collect}((\rho_1, \ldots, \rho_n), n) &= \mathit{collect}(\rho_1, n) \oplus \cdots \oplus \mathit{collect}(\rho_n, n) \end{aligned} \tag{8.29}$$

The annotation is something else

If the annotation is not a simple tree of regions, it may contain any construct from our annotation language. To retrieve the regions we need, we define a function called *gatherConstraintsTuple*. This function retrieves all unique regions from an annotations (local regions, externally bound variables and the regions from constraint sets) and returns them as a tuple. The exact implementation of this function can be found in the analysis implementation. The output of *gatherConstraintsTuple* is a tuple of regions, which we know how to handle. The code for this method can be found in the implementation [34].

$$\frac{}{\{\psi \mapsto n\}[\psi := a] \longrightarrow \{\psi \mapsto n\}[\psi := \mathit{gatherConstraintsTuple}(a)]} \tag{8.30}$$

8.3.3 Instantiation

At an instantiation the type argument α is substituted for τ. If the type variable α is instantiated in a polymorphic region sort or polymorphic sort, then P_Γ and Φ_Γ are used to instantiate the sort.

$$\frac{}{\forall\alpha.a\ \{\tau\} \longrightarrow a[\alpha := \tau]} \tag{8.31}$$

8.3.4 Projection

When the sub-annotation in the projection evaluates to a n-tuple, the targeted value can be projected out.

$$\frac{0 \leq i < n}{\pi_i((a_0, \ldots, a_i, \ldots, a_{n-1})) \longrightarrow a_i} \tag{8.32}$$

8.3.5 Addition

The rule for addition simply uses the definition of '$\oplus$'. It is assumed that the addition operator only occurs between annotations of sort C.

$$\frac{C_1 \oplus C_2 = C_3}{C_1 \oplus C_2 \longrightarrow C_3} \tag{8.33}$$

8.3.6 Top & bottom

Top can in some cases be evaluated and broken down. Note that when an abstraction is broken down, the variable is captured in the constraint set of top.

$$\frac{}{\top[c : s_1 \mapsto s_2] \longrightarrow \lambda\psi : s_1.\top[c \oplus \{\psi \mapsto \infty\} : s_2]} \tag{8.34}$$

$$\frac{}{\top[c : (s_1, \ldots, s_n)] \longrightarrow (\top[c : s_1], \ldots, \top[c : s_n])} \tag{8.35}$$

$$\frac{}{\top[c : \forall\alpha.s] \longrightarrow \forall\alpha.\top[c : s]} \tag{8.36}$$

$$\frac{}{\top[c : C] \longrightarrow c} \tag{8.37}$$

$$\frac{}{\top[c : ()] \longrightarrow ()} \tag{8.38}$$

Bottom can be broken apart in a similar manner.

$$\frac{}{\bot[s_1 \mapsto s_2] \longrightarrow \lambda\psi : s_1.\bot[s_2]} \tag{8.39}$$

$$\frac{}{\bot[(s_1, \ldots, s_n)] \longrightarrow (\bot[s_1], \ldots, \bot[s_n])} \tag{8.40}$$

$$\frac{}{\bot[\forall\alpha.s] \longrightarrow \forall\alpha.\bot[s]} \tag{8.41}$$

$$\frac{}{\bot[C] \longrightarrow \{\}} \tag{8.42}$$

$$\frac{}{\bot[()] \longrightarrow ()} \tag{8.43}$$

8.3.7 Join

The join over two constraint sets uses the definition of join from section 8.2.2. If it not possible to derive the least element then the join is not evaluated. The join operator is symmetric, so for any rule in the form of $a_1 \sqcup a_2 \longrightarrow a_3$ the symmetric case $a_2 \sqcup a_1 \longrightarrow a_3$ holds as well.

$$\frac{C_1 \sqcup C_2 = C_3}{C_1 \sqcup C_2 \longrightarrow C_3} \tag{8.44}$$

$$\frac{}{\Lambda\psi.a_1 \sqcup \Lambda\psi.a_2 \longrightarrow \Lambda\psi.(a_1 \sqcup a_2)} \tag{8.45}$$

$$\frac{}{\forall\alpha.a_1 \sqcup \forall\alpha.a_2 \longrightarrow \forall\alpha.(a_1 \sqcup a_2)} \tag{8.46}$$

$$\frac{}{a_1\{\tau\} \sqcup a_2\{\tau\} \longrightarrow (a_1 \sqcup a_2)\ \{\tau\}} \tag{8.47}$$

$$\frac{}{(a_1, \ldots, a_n) \sqcup (b_1, \ldots, b_n) \longrightarrow (a_1 \sqcup b_1, \ldots, a_n \sqcup b_n)} \tag{8.48}$$

$$\frac{}{() \sqcup a \longrightarrow ()} \tag{8.49}$$

$$\frac{}{(a_1 \oplus a_2) \sqcup (a_3 \oplus a_4) \longrightarrow (a_1 \sqcup a_3) \oplus (a_2 \sqcup a_4)} \tag{8.50}$$

$$\frac{}{(a_1 \setminus \rho) \sqcup (a_2 \setminus \rho) \longrightarrow (a_1 \sqcup a_2) \setminus \rho} \tag{8.51}$$

$$\frac{}{\bot[s] \sqcup a \longrightarrow a} \tag{8.52}$$

$$\frac{}{\top[c_1 : C] \sqcup \top[c_2 : C] \longrightarrow \top[c_1 \sqcup c_2 : s]} \tag{8.53}$$

If an annotation is joined with top, we must absorb the information from said annotation into top. For this purpose we use a method called *gatherConstraints*, which converts the annotation into a constraint set with all regions from the annotation a mapped to unbounded. This function is similar to *gatherConstraintsTuple* from Section 8.3.2. The code for these methods can be found in the implementation [34].

$$\frac{}{\top[c : s] \sqcup a \longrightarrow \top[c \sqcup \mathit{gatherConstraints}(a) : s]} \tag{8.54}$$

A projection on a join of tuples is moved outward.

$$\frac{}{\pi_i[a \sqcup b] \longrightarrow \pi_i[a] \sqcup \pi_i[b]} \tag{8.55}$$

The reason for normalizing projections in this manner is that taking the join of two projections may form an ill-sorted annotation, for example '$\lambda\phi : ((), ()).\pi_0[\phi] \sqcup \pi_0[((), \{\rho \mapsto 1\})]$' would be evaluated to '$\lambda\phi : ((), ()).\pi_0[\phi \sqcup ((), \{\rho \mapsto 1\})]$'. This annotation is ill-sorted according to sorting rule 8.10. Evaluating the annotation this way makes sure we only have joins of projections, not the other way around. This is important for fixpoints.

An application on a join is also moved outward, because taking the join of applications requires either the body or argument to be equal. Checking this may be an expensive operation.

$$\frac{}{(a_1 \sqcup a_2)\langle a_3 \rangle \longrightarrow a_1\langle a_3 \rangle \sqcup a_2\langle a_3 \rangle} \tag{8.56}$$

9

Region Bound Inference

We now present the inference rules after which we will show some example derivations.

9.1 Higher ranked type and effect

The inferred annotations can be higher ranked. This however is somewhat masked as we do not embed our annotations in the terms of the underlying lambda language as Thorand does. Because the annotations are separate from the terms we will refer to them as term level and annotation level as presented in Figure 9.1. Note that sort assignment is required to deal with higher ranked annotations, because higher ranked type/sort derivation is undecidable [7].

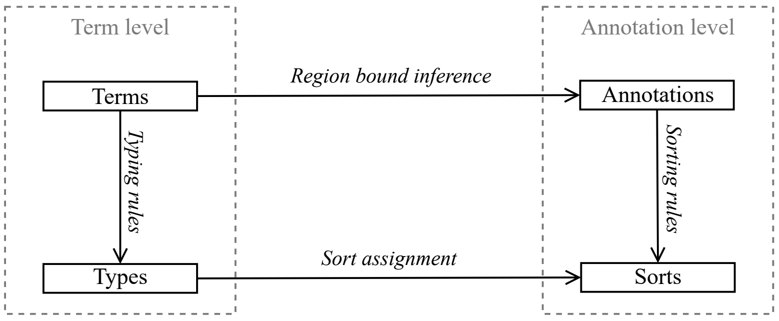

Figure 9.1: A representation of the analysis structure

9.2 Inference rules

For inferring the annotation of an expression we create the following inference rules. The notation '$\Delta; \mathcal{P}; \Gamma \vdash e \hookrightarrow a \mathbin{\&} \gamma$' denotes 'Under effect environment Δ, region environment $\mathcal{P}$ and datatype environment Γ the expression e has the annotation a and the effect γ', where '$a \mathbin{\&} \gamma$' is syntactic sugar for '(a, γ)'. Even though the effect will always be of the constraint set sort C it can still contain all of the annotation constructs.

The effect environment maps term variables to annotation variables. The region environment maps regions to region variables or local regions. The region environment is required because region variables are bound to annotation level abstractions, while local regions are defined by a letregion.

Integers Values are stored in a region by an '**at**' expression, not by the constant itself. This means that an integer value has the unit annotation and no effect. Note that in a practical scenario integers will not be assigned to regions, as they are stack allocated by default.

$$\frac{n \in \mathbb{N}}{\Delta; \mathcal{P}; \Gamma \vdash n \hookrightarrow () \ \& \ \bot} \quad \text{(integer)}$$

Variable The annotations of variables are stored in the environment. Reading from a variable has no effect.

$$\frac{x \mapsto a \in \Delta}{\Delta; \mathcal{P}; \Gamma \vdash x \hookrightarrow a \ \& \ \bot} \quad \text{(variable)}$$

Abstraction For an abstraction $\lambda x : \tau_x.e$ with the type $\tau_x \to \tau_r$, we extend the environments with an annotation variable ψ and a region variable ϕ. The annotation and effect of the body are stored in a tuple, which is wrapped in the abstractions for the annotation variable and region variable. Sort and region assignment is used to assign the sort for the annotation variable and return regions.

$$\frac{\Delta, x \mapsto \psi; \mathcal{P}, \rho \mapsto \phi; \Gamma \vdash e \hookrightarrow a \ \& \ \gamma}{\Delta; \mathcal{P}; \Gamma \vdash \lambda[\rho]x : \tau_x.\ e \hookrightarrow \Lambda\psi : \Phi_\Gamma(\tau_x).\Lambda\phi : P_\Gamma(\tau_r).(a, \gamma) \ \& \ \bot} \quad \text{(abstraction)}$$

Application For application it is first determined what the annotation and effect of the function f and the argument x are. The function annotation is an annotation level lambda, which when given the return region and annotation of the argument returns a tuple of the annotation and effect of the result.

$$\frac{\Delta; \mathcal{P}; \Gamma \vdash f \hookrightarrow a_f \ \& \ \gamma_f \qquad \Delta; \mathcal{P}; \Gamma \vdash x \hookrightarrow a_x \ \& \ \gamma_x \qquad (a_r, \gamma_r) = a_f\langle a_x\rangle\langle\phi\rangle \qquad \rho \mapsto \phi \in \mathcal{P}}{\Delta; \mathcal{P}; \Gamma \vdash f[\rho]\langle x\rangle \hookrightarrow a_r \ \& \ \gamma_f \oplus \gamma_x \oplus \gamma_r} \quad \text{(application)}$$

If-then-else The ITE takes the join of the annotations of the branches. The join of the effects of the branches is sound because only *one* of the two branches will be executed. This join represents subtyping, because either branch can have an annotation that is less restrictive than the annotation of the ITE.

$$\frac{\Delta; \mathcal{P}; \Gamma \vdash x \hookrightarrow a_c \ \& \ \gamma_c \qquad \Delta; \mathcal{P}; \Gamma \vdash e_1 \hookrightarrow a_t \ \& \ \gamma_t \qquad \Delta; \mathcal{P}; \Gamma \vdash e_2 \hookrightarrow a_f \ \& \ \gamma_f}{\Delta; \mathcal{P}; \Gamma \vdash \textbf{if } x \textbf{ then } e_1 \textbf{ else } e_2 \hookrightarrow a_t \sqcup a_f \ \& \ \gamma_c \oplus (\gamma_t \sqcup \gamma_f)} \quad \text{(if-then-else)}$$

Let For a let-expression it is first determined what the annotation and effect of the defined variable x are. The annotation of x is added to the environment to determine the type of the subexpression.

$$\frac{\Delta; \mathcal{P}; \Gamma \vdash e_x \hookrightarrow a_x \ \& \ \gamma_x \qquad \Delta, x \mapsto a_x; \mathcal{P}, e; \Gamma \vdash_b \hookrightarrow a_b \ \& \ \gamma_b}{\Delta; \mathcal{P}; \Gamma \vdash \textbf{let } x = e_x \textbf{ in } e_b \hookrightarrow a_b \ \& \ \gamma_x \oplus \gamma_b} \quad \text{(let)}$$

Letregion The region environment is extended with the local region ρ. At a letregion the local region is removed from the constraint set using the restrict operation. Note that the bound on this region should be stored somewhere at this point, but this is left as an implementation detail.

$$\frac{\Delta; \mathcal{P}, \rho \mapsto \rho; \Gamma \vdash e \hookrightarrow a \ \& \ \gamma \qquad \gamma' = \gamma \setminus \rho}{\Delta; \mathcal{P}; \Gamma \vdash \textbf{letregion } \rho \textbf{ in } e \hookrightarrow a \ \& \ \gamma'} \tag{letregion}$$

At The region that is allocated to may be a region variable, so it is looked up in the region environment. Note that the region environment may return a local region as well. The subexpression is evaluated and the bound on the size of ρ is increased by one.

$$\frac{\Delta; \mathcal{P}; \Gamma \vdash e \hookrightarrow a \ \& \ \gamma \qquad \rho \mapsto \phi \in \mathcal{P}}{\Delta; \mathcal{P}; \Gamma \vdash e \textbf{ at } \rho \hookrightarrow a \ \& \ \{\phi \mapsto 1\} \oplus \gamma} \tag{at}$$

Tupling & projection The tupling and projection rules are straightforward as well. The annotations are tupled in the tupling rule and projected out in the projection rule. The effects of all the elements of the N-tuple are combined with the addition operator.

$$\frac{\Delta; \mathcal{P}; \Gamma \vdash x_0 \hookrightarrow a_0 \ \& \ \gamma_0, \ldots, \Delta; \mathcal{P}; \Gamma \vdash x_n \hookrightarrow a_n \ \& \ \gamma_n}{\Delta; \mathcal{P}; \Gamma \vdash (x_0, \ldots, x_n) \hookrightarrow (a_0, \ldots, a_n) \ \& \ \gamma_0 \oplus \cdots \oplus \gamma_n} \tag{tupling}$$

$$\frac{\Delta; \mathcal{P}; \Gamma \vdash e \hookrightarrow a \ \& \ \gamma}{\Delta; \mathcal{P}; \Gamma \vdash \pi_i(e) \hookrightarrow \pi_i(a) \ \& \ \gamma} \tag{projection}$$

Quantification and instantiation Quantifications and instantiations are mirrored in the annotation language.

$$\frac{\Delta; \mathcal{P}; \Gamma \vdash e \hookrightarrow a \ \& \ \gamma}{\Delta; \mathcal{P}; \Gamma \vdash \forall \alpha.e \hookrightarrow \forall \alpha.a \ \& \ \gamma} \tag{quantification}$$

$$\frac{\Delta; \mathcal{P}; \Gamma \vdash e \hookrightarrow a \ \& \ \gamma}{\Delta; \mathcal{P}; \Gamma \vdash e \ \{\tau\} \ \hookrightarrow a \ \{\tau\} \ \& \ \gamma} \tag{instantiaton}$$

9.3 Example derivations

We will now do some example derivations to demonstrate the derivation rules.

Derivation of 'id' The derivation of `id` is quite simple. The resulting annotation clearly shows that the annotation of the parameter ψ is returned. The effect of the function is '$\bot$'. Both annotation and effect are correct, because `id` does not have any effect.

$$\cfrac{\cfrac{\cfrac{x \mapsto \psi \in \{x \mapsto \psi\}}{\{x \mapsto \psi\}; \{\hat{\rho} \mapsto \phi\}; \Gamma \vdash x \hookrightarrow \psi \ \& \ \bot \qquad \textit{returntype} = \alpha}\ \textsc{var}}{\{\}; \{\}; \Gamma \vdash \lambda[\hat{\rho}]x : \alpha.x \hookrightarrow \Lambda\psi : \Phi_\Gamma(\alpha).\Lambda\phi : P_\Gamma(\alpha).(\psi, \bot) \ \& \ \bot}\ \textsc{abs}}{\{\}; \{\}; \Gamma \vdash \forall \alpha.\lambda[\hat{\rho}]x : \alpha.\ x \hookrightarrow \forall \alpha.\Lambda\psi : \Phi_\Gamma(\alpha).\Lambda\phi : P_\Gamma(\alpha).(\psi, \bot) \ \& \ \bot}\ \textsc{qnt}$$

Derivation of '\$' The annotations (and derivations) grow quite quickly as can be seen with the derivation of `($)`. Because the derivation tree is quite large, it can be found in Appendix B. The derived annotation of `($)` is presented in Equation 9.1.

$$\begin{aligned}
&\forall\alpha.\forall\beta.\\
&\quad\Lambda\psi_f : \Phi_\Gamma(\alpha \to \beta).\Lambda\phi_f : P_\Gamma(\alpha \to \beta).\\
&\qquad(\Lambda\psi_x : \Phi_\Gamma(\alpha).\Lambda\phi_x : P_\Gamma(\beta).\\
&\qquad\quad(\pi_0[\psi_f\langle\psi_x\rangle\langle\phi_x\rangle]\\
&\qquad\quad,\pi_1[\psi_f\langle\psi_x\rangle\langle\phi_x\rangle])\\
&\qquad,\bot)
\end{aligned} \tag{9.1}$$

If `id` is applied to `($)` it will evaluate to the annotation and effect seen in 9.2, which is equivalent to `id`. This is exactly the annotation that is expected from `($) id`.

$$\forall\alpha.\Lambda\psi : \Phi_\Gamma(\alpha).\Lambda\phi : P_\Gamma(\alpha).(\psi, \bot) \tag{9.2}$$

9.3.1 Versus ML Kit

We will now compare our analysis with some examples presented in Vejlstrups work [22]. In a chapter about the incompleteness of his algorithm Vejlstrup presents two cases where his algorithm delivers suboptimal results.

For the program (9.3), Vejlstrup derives a bound of two on the return region of the function argument. Using our inference rules we can define a bound of one on that same region. This might seem like a small difference, however, Vejlstrup advises to use $K = 1$, which would make the region unbounded in his analysis and bounded in ours.

$$\begin{aligned}
&\lambda[\rho_1]f.\ \textbf{let}\ x = \textbf{if}\ \mathit{true}\ \textbf{then}\ 1\ \textbf{else}\ f[\rho_1]\ 1\\
&\qquad\quad\ \textbf{in}\ (\textbf{if}\ \mathit{true}\ \textbf{then}\ (\lambda[\rho_2]y.1\ \textbf{at}\ \rho_2)\ \textbf{else}\ f)
\end{aligned} \tag{9.3}$$

For the program (9.4), Vejlstrup derives that the return region of the function argument must be unbounded (due to cyclic arrow effects), regardless of which value the parameter K is set to. Note Our inference rules can derive that the bound must be two on said region, which makes the region bounded.

$$\begin{aligned}
&(\lambda[\rho_1]f.\ \textbf{let}\ g = \lambda[\rho_2]y.f[\rho_2]\ y\\
&\qquad\quad\ \textbf{in}\ (\textbf{if}\ \mathit{true}\ \textbf{then}\ g\ \textbf{else}\ f))\ (\lambda[\rho_3]x.1\ \textbf{at}\ \rho_3)
\end{aligned} \tag{9.4}$$

9.4 Fixpoints

For recursive abstractions a recursive annotation is derived as well. For some recursive function f the effect environment is extended with an annotation variable ψ_f. After deriving the annotation of f, ψ_f may occur in the body of the annotation of f. The type of f is used to assign the sort to the fixpoint. Note that the sort of the derived annotation a is equal to the sort assigned by $\Phi_\Gamma(\tau_f)$.

$$\frac{\Delta, f \mapsto \psi_{fix}; \mathcal{P}; \Gamma \vdash \lambda[\rho]x : \tau_x.\ e \hookrightarrow a\ \&\ \gamma}{\Delta; \mathcal{P}; \Gamma \vdash \mu f : \tau_f.\lambda[\rho]x : \tau_x.\ e \hookrightarrow fix\ \Phi_\Gamma(\tau_f)\ .\ a} \quad \text{(recursive abs.)}$$

An example of recursive annotation can be seen in Equation 9.5. Note that a may contain the variable ψ_{fix}, which is bound to the fixpoint.

$$fix\ s.\ a \tag{9.5}$$

To compute the annotation of f we must do fixpoint iteration. The initial state of the fixpoint iteration is $\mu_0 = \bot[s]$. The next state can be computed by transforming the fixpoint into an abstraction and passing the state to it.

$$\mu_{i+1} = (\Lambda\psi_{fix} : s.\ a)\ \mu_i \tag{9.6}$$

Fixpoint iteration continues until $\mu_{i+1} = \mu_i$ holds. It is unknown how many iterations are required, so we introduce a second analysis parameter F, where F is some positive integer. If $\mu_{F+1} \neq \mu_F$ iteration is halted and the top annotation $\top[c : s]$ is returned. In this case fixpoint iteration has *failed.* The constraint set c maps all variables in scope to unbounded. The sort s is the sort from the fixpoint.

Example

Say the annotation seen in Equation 9.7 is derived from some program.

$$fix\ (() \mapsto P \mapsto ((), C)).\ \Lambda\psi : ().\Lambda\phi : P.(() \sqcup \pi_0[\psi_{fix}\langle\psi_a\rangle\langle\phi\rangle], \{\phi \to 1\}) \tag{9.7}$$

The fixpoint iteration with $\bot$ annotated with the fixpoint sort. The next state is determined by Equation 9.6. After two iterations we have that $\mu_2 = \mu_1$, which means that we can terminate the fixpoint iteration.

$$\begin{aligned}
\mu_0 &= \bot[() \mapsto P \mapsto ((), C)] \\
\mu_1 &= (\Lambda\psi_{fix}.\Lambda\psi.\Lambda\phi.(() \sqcup \pi_0[\psi_{fix}\langle\psi_a\rangle\langle\phi\rangle], \{\phi \to 1\}))\langle\bot[() \mapsto P \mapsto ((), C)]\rangle \\
&= \Lambda\psi.\Lambda\phi.((), \{\phi \to 1\}) \\
\mu_2 &= (\Lambda\psi_{fix}.\Lambda\psi.\Lambda\phi.(() \sqcup \pi_0[\psi_{fix}\langle\psi_a\rangle\langle\phi\rangle], \{\phi \to 1\}))\langle\Lambda\psi.\Lambda\phi.((), \{\phi \to 1\})\rangle \\
&= \Lambda\psi.\Lambda\phi.((), \{\phi \to 1\})
\end{aligned} \tag{9.8}$$

9.4.1 Top

If fixpoint iteration does not terminate within F steps we must fall back to top. We use the method *gatherConstraints* to retrieve all variables used in a and map them to unbounded. Note that both region variables as well as annotation variables are captured in the resulting constraint set.

$$\frac{fix\ s.\ a}{\top[gatherConstraints(a) : s]} \tag{9.9}$$

9.5 Datatypes

We now return to datatypes. As one may have noticed, there are no datatypes in the annotation language. This is because datatypes are converted to tuples of their fields. The constructors and destructors are converted to annotations before analyzing a module. These annotations are put in the datatype environment Γ. The example datatype `Foo` presented in Figure 9.2 will be used repeatedly throughout the examples.

Figure 9.2: Simple datatype

```
data Foo a b = Bar a b
             | Baz a
```

9.5.1 Constructors

Constructors in Haskell are similar to functions. A constructor of n fields takes n arguments. A datatype constructor can be converted to a tuple of the fields wrapped in abstractions and quantifications.

For the tuple of fields we follow a similar procedure to assigning sorts as in Section 7.3. Each constructor becomes a tuple of its fields, each datatype a tuple of its constructors. All the fields that are not part of the constructor are set to $\bot[s]$, where s is the sort of the corresponding field. All the fields that are part of the constructor are bound to an abstraction. Finally, the constructor is wrapped in quantifications for the type arguments of the datatype.

During analysis the constructor can simply be looked up in the datatype environment Γ. The (type) arguments are applied by type instantiation and applications in the term.

$$\frac{\Delta; \mathcal{P}; \Gamma \vdash D \mapsto a \in \Gamma}{\Delta; \mathcal{P}; \Gamma \vdash D \hookrightarrow a} \tag{9.10}$$

Example

In the example datatype seen in Figure 9.2 there are two constructors. Using the rules as described the constructors are mapped to the annotations in Equation 9.11.

$$\begin{aligned} Bar &\mapsto \forall\alpha.\forall\beta.\Lambda a : \Psi\langle\alpha\rangle.\Lambda b : \Psi\langle\beta\ []\rangle.((a, b), (\bot[\Psi\langle\alpha\rangle])) \\ Baz &\mapsto \forall\alpha.\forall\beta.\Lambda a : \Psi\langle\alpha\rangle.((\bot[\Psi\langle\alpha\rangle], \bot[\Psi\langle\beta\ []\rangle]), (a)) \end{aligned} \tag{9.11}$$

When we derive the annotation for some term that contains the constructor Bar, it is looked up in the environment Γ. In Equation 9.12 the type arguments are initialized with Int and an integer and a variable are passed, which results in the annotation on the right hand side of the arrow.

$$\lambda c : ().Bar\ \{Int\}\{Int\}\ 5\ c \hookrightarrow\ \Lambda c : ().(((), c), \bot[()]) \tag{9.12}$$

9.5.2 Case & destructors

At a case expression the fields of the datatype have to be bound to variable. A datatype is broken apart using destructors. A destructor is the inverse operation of a constructor. Destructors project out the fields from the datatype and bind them to the corresponding variables.

Destructors are generated for every field of every constructor of every datatype. A destructor for the j-th field in the i-th constructor is the j-th element of the i-th element of the datatype tuple. Note that k_i represents the number of fields of the constructor, which may vary per constructor in a datatype.

$$\frac{\Delta, y_{i,1} \mapsto \psi_{i,1}\langle\psi_x\rangle, \ldots, y_{i,k_i} \mapsto \psi_{i,k_i}\langle\psi_x\rangle; \mathcal{P}; \Gamma \vdash\mapsto a_i \qquad \forall i.\ 0 < i \leq n}{\Delta, x \mapsto \psi_x; \mathcal{P}, \mathbf{;}\Gamma \vdash \mathit{casex} : D \ \mathbf{of}\ D_1\ \bar{y}_1 \rightarrow e_1; \ldots; D_n\ \bar{y}_n \rightarrow e_n \hookrightarrow a_1 \sqcup \cdots \sqcup a_n} \tag{9.13}$$

$$\psi_{i,j} = \textit{The destructor for the j-th field in the i-th constructor}$$
$$\textit{of datatype D in datatype environment } \Gamma$$

Example

The destructors for the example datatype seen in Figure 9.2 are presented in Equation 9.14. Note that the argument a has the sort of the datatype. Each field is projected out of the datatype tuple.

$$\begin{aligned} \mathit{Bar}\ x\ y \mapsto \{&x \mapsto \forall\alpha.\forall\beta.\Lambda a : ((\Psi\langle\alpha[]\rangle, \Psi\langle\beta[]\rangle), (\Psi\langle\alpha[]\rangle)).\pi_0[\pi_0[x]], \\ &y \mapsto \forall\alpha.\forall\beta.\Lambda a : ((\Psi\langle\alpha[]\rangle, \Psi\langle\beta[]\rangle), (\Psi\langle\alpha[]\rangle)).\pi_1[\pi_0[x]]\} \\ \mathit{Baz}\ z \mapsto \{&z \mapsto \forall\alpha.\forall\beta.\Lambda a : ((\Psi\langle\alpha[]\rangle, \Psi\langle\beta[]\rangle), (\Psi\langle\alpha[]\rangle)).\pi_0[\pi_1[x]]\} \end{aligned} \tag{9.14}$$

9.5.3 Simple recursive datatypes

We currently do not support this class of datatypes, however we did implement constructor and destructor annotations for lists by hand. The datatypes not supported in this category are handled in the same manner as in Section 9.5.4.

The by hand created constructor annotation for `(:)` produces an annotation for the new element as with simple non recursive datatypes. We then take the join of this annotation with the list argument (Equation 9.15). The destructor for `(:)` projects out the element of the list and returns the entire annotation as the recursive list (Equation 9.16). The join makes sure that the value with the highest constraints in the list is always projected out by the destructor.

$$(:)\ \mapsto \forall\alpha.\Lambda x : \Phi_\Gamma(\alpha).\Lambda xs : s.((), (x, ())) \sqcup xs \tag{9.15}$$

$$\begin{aligned} (:)\ x\ xs \mapsto \{\ &x \mapsto \forall\alpha.\Lambda a : s.\ \pi_0[\pi_1[\pi_0[a]]], \\ &xs \mapsto \forall\alpha.\Lambda a : s.\ s\} \end{aligned} \tag{9.16}$$

$$s = (((), (\Phi_\Gamma(\alpha), ())))$$

9.5.4 Complex datatypes

We are unable to analyze some datatypes. Every datatype that we cannot analyze will simply have the unit sort. The constructor will return an annotation unit. The destructor will create a top annotation with the sort of the matched field. Note that in case of recursive datatypes the sort of the recursive position is also a unit. These datatypes are also assigned a single nested region to put the bounds on from the fields.

10 The Helium Compiler

The Helium compiler is focused primarily on learning Haskell [35]. It does so by generating precise type error messages, for instance by either leaving out more complicated details (e.g. overloading) or suggesting fixes. The compiler is built and maintained at the University of Utrecht.

Recently work has started on a region based memory management system [11]. Our work will extend upon the region inference system to infer the sizes of the inferred regions.

10.1 The pipeline

The new Helium backend[24] compiles Haskell into Core, Core into Iridium and Iridium into LLVM. Each intermediate language runs its own analyses and has its own strengths.

10.2 Core

Core is a typed lambda language. The Core language is a simplified version of Haskell, which makes it easier to create analyses for.

10.2.1 Analyses

There are currently 9 unique passes on the Core language. Some passes are run more than once. The passes are run in the order of appearance.

Rename Make all variable names unique.

Saturate Saturates all calls to constructors and external calls, for example, '`(:) 1`' will be saturated and be turned into '`λx.(:) 1 x`'.

LetSort Sort recursive and non recursive bindings. The goal is to reduce the number of binding in a single let-expression by splitting let bindings that do not depend on each other.

LetInline Inlines let binds into the body if the variable is only used once (the result of the thunk is not shared). Removes unused let bindings. The analysis is run twice in a row to increase the accuracy.

Normalize Add let declarations for non-trivial sub expressions, e.g. '`f (g x)`' becomes '`let y = g x in f y`'.

Strictness Propagates strictness information and changes lazy let bindings to strict let bindings. The analysis is run twice in a row to increase the accuracy.

RemoveAliases Removes aliases of variables, e.g. `let x = y in f x` becomes 'f y'. Removes strict let bindings if `y` is already evaluated strictly. The pass also flattens match constructs that match repeatedly on the same variable.

ReduceThunks Reduce the number of thunks that will be created. It does so by evaluating cheap expressions strictly, for example constructors and literals can be evaluated strictly without changing the semantics.

Lift Lift lambdas and non-strict let declarations to top level.

Strictness Strictness is run again as a final pass to propagate the strictness information again as it might have changed in the previous set of passes.

10.3 Iridium

Iridium is a language with a functional type system, but an imperative flow. Memory management is explicit, which makes it better suited for lower level analyses (i.e. region(size) inference). Laziness is also explicit. Thunks are created and evaluated by instructions in the Iridium language. There are things which are not represented in Core (e.g. thunk allocation) that do require memory management.

10.3.1 Modules

Just like in Haskell, code in Iridium is divided into modules. Each module has its own file. Modules can import other modules (or parts of those modules). Module imports cannot be recursive.

Compilation is done on a module-by-module basis. If we want to transfer an annotation between modules it must be written to the Iridium file. Parsing only happens when recompiling a subset of the modules of a program, otherwise the data is read from a cache.

Binding groups

Methods and datatypes can be put into binding groups. A binding group represents a group of methods/datatypes that depend on each other. The set of all binding groups in a module form a tree of dependencies which can be traversed topologically during analysis.

10.3.2 Methods

Methods behave similarly to as functions do in Haskell (and Core). An method can be partially applied, passed around and/or returned as an argument. A method header for Helium Prelude's id [36] can be found in Equation 10.1.

A method is annotated with a unique identifier (`@id/@Prelude.id`) followed by its type (`forall a.  a -> a`). Method do not have to be exported, in which case the `export_as [name]` is eluded. After the dollar sign follows the list of arguments annotated with their types. After the colon follows the return type.

A method is also annotated with the regions inferred by the region inference pass [11]. After the return type follows an at-sign and the return regions. A method may also require additional regions, which is a flat tuple that can occur before the list of arguments.

Finally the method is annotated with analysis annotations. These annotations are required to do partial recompilation. The annotations from these analyses are put between square brackets and flagged with the analysis name.

Figure 10.1: Iridium method header

```
export_as @id define @Prelude.id: { (forall a. a -> a) }
    $ (forall a, b: !a): a @ (rho_1,rho_2)[trampoline]
    [region: ...]
    [regionsize: ...] { ... }
```

10.3.3 Blocks

Every method is divided into blocks. A block is simply a label and an instruction. Jump and case instructions jump between these blocks. A method always starts execution at the 'entry' block. Examples of blocks, jumps and cases can be found in Figure 10.3.

10.3.4 Instructions

Iridium has 5 non-terminal instructions.

let *id expr next*: Assigns the expression to the identifier and evaluates the 'next' instruction.

letalloc *binds next*: Allocates memory for the bindings and assigns them to their identifiers and evaluates the 'next' instruction.

newregion ρ *next*: Defines a new region ρ and evaluates the 'next' instruction.

releaseregion ρ *next*: Allows region ρ to be deallocated and evaluates the 'next' instruction.

match Assigns fields from constructors or tuples to local variables.

Iridium also has four terminal instructions

jump *id*: Jumps unconditionally to the block with identifier *id*.

case *local case*: Jumps to the block that matches the constructor or integer.

return *local*: Returns the local variable and passes control back to the caller.

unreachable Denotes that this instruction should not be reached. Can be used after a call to a function like 'error'.

Binds

Binds in `letalloc` instructions allocate memory for thunks and values. There are 4 kinds of allocation targets: functions, thunks, tuples and constructors.

Functions and thunks are annotated with their type followed by a dollar sign. After the dollar sign come the thunk regions followed by another dollar sign. After that dollar sign follow type and regular arguments. The result value of a thunk must be stored in a region, this region is denoted by '`@ rho_n`'.

Tuples are handled separately from the other datatypes. Tuples are annotated with their arity followed by a dollar sign and its arguments. The tuple is stored in a region, this region is denoted by '`@ rho_n`'. Constructors are handled similarly. Instead of an arity they are annotated with a constructor identifier and the type of the constructor.

Figure 10.2: Iridium bind examples

```
; Function allocation
letalloc %f1 = function @Prelude.id[1]: (forall a. !a -> a)
            $ ((), (), (rho_global, ()))
            $ ({a})

; Thunk allocation
letalloc %f2 = thunk %fTuple: !((,) Int Int -> (,) Int Int)
            $ ((), (rho_4, (rho_3, rho_2, (), rho_1, rho_0, ())))
            $ (x: ((,) Int Int)) @ rho_4

; Tuple allocations
letalloc %t = tuple 2
            $ ({Int}, {Int}, %a: Int, %b: Int) @ rho_4

; Constructor allocation
letalloc %l = constructor @":": (forall a. a -> [a] -> [a])
            $ ({a}, %x: a, %xs: ([a])) @ rho_0
```

10.3.5 Expressions

Expressions behave similarly to Haskell and Core. They do not contain a 'next' statement as some instructions do.

literal A literal value, either an integer, float or string.

call $@\mathit{name}[n]\ :\ \tau_f\ \ \rho_a\ \ \left(\overline{\tau_a \mid \mathit{var}}\right)\ \ @\ \rho_r$: Calls a method with additional region argument(s) ρ_a, type and value arguments $\left(\overline{\tau_a \mid \mathit{var}}\right)$ and the return variable(s) ρ_r. The method is annotated with its arity n.

instantiate *local* $\bar{\tau}$: Instantiate a list of types in the polymorphic local variable.

eval *var*: Evaluates a value to weak head normal form (WHNF) and returns it or simply returns the value if it is already in WHNF.

var *var*: Gets the value of the variable and does not evaluate it.

cast *local* τ: Casts a value to a (possibly different) type.

castthunk *local*: Cast a strict type to a non-strict type, i.e. converts $!\tau$ to τ.

phi $\overline{\mathit{branch}}$: Gets a value based on the previous block. The phi node is explained in more detail in Section 10.3.6.

primitive Calls some primitive instruction, such as integer addition. The primitive expression call should be fully saturated.

undefined τ: Denotes some value of type τ that is not defined and returns some arbitrary value. A program may crash when using a value produced by undefined. Undefined can be used in optimization to represent a value that is not used. This expression is not the Haskell function `undefined` as it does not raise an error.

seq $a\ b$: Marks a dependency between the value of a and b. Note that it does not evaluate a. Ignores the value of a and returns the value of b. Seq can be used to compile Haskell functions like `seq`.

10.3.6 Static Single-Assignment form

Variables in Iridium can only be assigned to once. This restriction makes it easier to reason about programs. In case multiple assignments are required, it is made clear by using a phi-node. LLVM is also in SSA form, which makes it easier to translate Iridium into LLVM.

Phi nodes

Phi nodes allow assigning to a variable based on the previously executed block. Consider the Haskell expression '`let x = case a of True -> 1; False -> 2`'.

Figure 10.3: Iridium code for '`let x = case a of True -> 1; False -> 2`'

```
entry:
    case %a: !bool constructor (
        @True[0]: Bool to branchtrue,
        @False[0]: Bool to branchfalse)
branchtrue:
    %y1 = literal int 1
    jump end
branchfalse:
    %y2 = literal int 2
    jump end
end:
    %x = phi (
        branchtrue  => %y1: !Int,
        branchfalse => %y2: !Int)
```

As can be seen in the generated Iridium code in Figure 10.3, the variable `%x` is assigned to by a phi node. If the variable `%a` is a true the program evaluates the 'branchtrue' block and sets `%y1` to the integer 1. The program then jumps to the 'end' block. In the end block the phi node knows that the previously executed block is 'branchtrue', so it assigns `%y1` to `%x`. If `%a` equals 'False' the program would evaluate the other block and set `%x` to `%y2`.

10.3.7 Analyses

Currently there are four analyses run on Iridium. The analyses are run in the order of appearance.

DeadCode Removes all unreachable code.

TailRecursion Transforms tail recursive functions into loops.

Region The work done by De Wolff [11]. Analyses the program and inserts `newregion` and `releaseregion` instructions. Annotates `letalloc` constructs with regions.

RegionSize The analysis presented in this thesis. Annotates the `newregion` instructions with a bound.

10.3.8 Thunks

For a deep dive into the workings of thunks in Iridium we recommend reading Section 8.6 from De Wolffs thesis [11].

10.4 LLVM

The Low-Level Virtual Machine (LLVM) project started as a research project in 2004 by the University of Illinois [37]. LLVM is more than just a language, it is a collection of tools which allows one to optimize code and generate CPU code for a wide variety of processors.

The LLVM language is imperative with SSA and explicit memory management.

10.4.1 Analyses

There is a large number of analyses and program transformations that the LLVM Core supports [38]. The Helium compiler uses a subset of these passes.

11 Implementation

We now focus on the actual analysis implementation. The region bound inference is implemented in the Helium compiler. The analysis is run on the Iridium intermediate language. Because we run the analysis after tail recursion, there may be loops in the Iridium code. The analysis derives how many times there may be something allocated to a region, not the number of bytes that may be allocated in each region.

11.1 Analysis on Iridium

Iridium is a language with a functional type system, but an imperative flow. This means that the analysis must be adapted slightly to deal with this difference in structure. Expressions can be dealt with in the same manner as the proposed analyses in Chapter 9. We take the join over the branches of phi nodes. We maintain a set of environments during the analysis. There are the ones from Chapter 8, but also an environment to store the annotation and effect from blocks.

Return *local*: Return the annotation of the local variable, has no effect.

Unreachable *local*: Return a bottom annotation with the sort matching the type of *local*. This instruction can never be reached, thus it will never do any allocations.

Jump *block*: Return the annotation and effect of the block.

Case *cases*: If a case matches on simple base types such as integers it is equivalent to nested if statements, which is equivalent to the join of their branches. If the case matches on a datatypes it is equivalent to the case construct presented in the source language, in which case we follow the inference rules presented in Section 9.5.

Match *fields next*: Equivalent to a case branch. Deconstructs a datatype and assigns each field to a variable as presented in Section 9.5. Add the annotation of each of the variables to the local environment.

Let *name expr next*: Analyze the expression and assign the annotation to the name in the environment. Return the annotation and effect of the *next* instruction.

LetAlloc *binds next*: Analyze all of the bindings and add them to the environment. Each **LetAlloc** assigns data to a region, returns the annotation and effect of the *next* instruction and adds the effects of the *binds*.

NewRegion *name next*: Can be ignored, return the annotation and effect of the *next* instruction.

ReleaseRegion *name next*: Return the annotation and effect of the *next* instruction. Reset the released region to zero using the restrict operator. In a program without loops this will have no effect. In a program with loops this will make sure the region bound does not grow by the loop.

11.1.1 Pipeline

Currently the analysis goes through a number of steps for each binding group. The annotations are checked with the sorting rules. It is also checked if the sort does not change during annotation evaluation and fixpoint iteration.

- Analyze method without local regions;
- Evaluate the annotation and solve the fixpoints;
- Re-analyze binding group with updated global environment for recursive positions. This does not create a new fixpoint for (mutually) recursive functions;
- Extract the effect from the annotation;
- Transform the program, remove unused regions;

11.2 Implementation details

Theory often does not match what occurs in practice. Our source language provides a model such that we can easily reason with it, but it not a programming language in itself. Things like method definitions are not taken into account in the source language, but must be dealt with in the analysis. There may also be cases in which we need to take special measures to make sure our analysis stays sound.

11.2.1 Thunk regions and value regions

Currently the analysis assumes that all thunks are evaluated. This means that at a bind we also put a bound on the value region of a value. Note that this is different from a strict semantic, because thunks are still allocated and their regions could have a different lifespan from the value regions.

11.2.2 Additional regions

A method in Iridium may require additional regions. Additional regions are required to compute intermediate values which are used to compute the result. The sort of the additional regions can be derived from the method header in the Iridium file. The additional regions are brought in scope by an additional lambda wrapped around the method annotation. Sort assignment for an Iridium method wraps the sort with a sort lambda with the sort of the additional regions.

11.2.3 Zero arity functions

For functions without arguments (also known as constants) we need to make a correction to the annotation. Analyzing Example 11.1 would result in allocations for the tuple. These allocations are usually handled by the return regions of such a method.

However, to remain consistent with the sorting rules, such a type does not have return regions. To fix this, the global region is applied to the return regions of such an annotation.

These constants may also be used in other methods, where it is expected that they do not have any additional regions. Because of this the global region is applied to any additional regions that constant may require, after which it is wrapped in an abstraction with unit sort. This abstraction is needed to remain consistent with other method annotations as the unit sort represents an empty additional region tuple.

Figure 11.1: Zero arity function

```
zeroArr :: (Int,Int)
zeroArr = (1,2)
```

11.2.4 Higher order application of local variables

A local region may be applied to a higher order function. In this case it cannot be predicted what kind of bound said higher order function will put on the local region. All such regions must be analyzed as unbounded.

11.2.5 Modules & parsing

For cross-module analysis, each method can be annotated with an annotation in the Iridium file. These files are stored in a cache during analysis for fast access and stored on disk after the compilation has completed. This allows for separate compilation where annotations for already compiled modules can be read from disk. This does require a parser and pretty printer for the annotation language which have both been implemented. Because parsing cannot tell the difference between 1-tuples and braces, we have prefixed all tuples with ‘TUP(...)’.

11.3 Fixpoints

In Iridium the fixpoint constrains a tuple of sorts and annotations instead of just one. We use the tuple to deal with loops in the method body and mutual recursion. This list will be referred to as the fixpoint tuple. Each element in the tuple may refer to another element in the tuple. These references may create a loop. A fixpoint is evaluated in n steps when none of the elements in the fixpoint change during the n-th iteration. Fixpoint iteration fails if the number of iterations exceeds the maximum defined by analysis parameter F.

Figure 11.2: Recursive variable definition

```
block1:
  %x = phi (block1 => literal int 5: !Int,
            block2 => %y2: !Int)
  jump block2
block2:
  %y = %x + 1
  jump entry
```

11.3.1 Block loops and recursive variables

Due to the fact that our region bound analysis comes after the tail recursion optimization programs may contain loops. Loops in Iridium are created by jumps between blocks. This however also has the effect that variables assigned to by phi nodes may have a recursive definition, which in turn gives a recursive annotation. This can be seen in Figure 11.2 `%x` is defined in terms of `%y`, which is defined in terms of `%x`.

Because of this we create a single fixpoint for variables and blocks. This fixpoint captures all recursive relations between blocks and variables.

11.3.2 Recursion and mutual recursion

For recursion and mutual recursion we create a fixpoint for a set of methods that are defined in terms of each other. In the case of mutually recursive we analyze the group of methods that is defined in terms of each other all at once.

11.3.3 Inlining

The local fixpoint tuple may become quite large. We can reduce the size of the fixpoint tuple by inlining all non-recursive elements. After inlining there are likely elements of the fixpoint tuple that are no longer used.

11.4 Datatypes

Before analyzing the methods in a module, we first create the datatype environment. The datatype environment contains the sort and region assignment for every datatype used in the module. The datatype environment also contains constructor and destructor annotations for every constructor of every datatype.

11.5 Testing

The code has been tested throughout the entirety of development. Starting with simple examples and working up to the more complex methods of the Helium Prelude.

The sorting rules have been a great aid in tracking down issues. Most bugs were found due to the incorrect sort of some method

11.6 Code

The implementation can be found in the Helium GitHub on the `region-size` branch [34]. The code for the implementation is contained in the `src/Helium/CodeGeneration/Iridium/RegionSize` folder.

12 Evaluation

In this chapter we will evaluate our solution and compare it to previous work. We will start by comparing our implementation to Vejlstrups work. We will then dive into the Helium Prelude [36] and look at some metrics. Finally, we will reflect on the current implementation.

12.1 Versus ML Kit

The examples from Section 9.3.1 that are problematic for the region size inference by Vejlstrup [22] are analyzed as presented in said section. In both cases our analysis outperforms Vejlstrups analysis. Note that Vejlstrup analysis does not have to deal with thunks because ML is strict.

Factorials & Fibonacci

In Vejlstrup's Thesis [22] it is claimed that ML's region inference will only allocate finite regions in the programs presented in Figure 12.1. Our analysis will always allocate integers on the stack. If we produce custom annotations for the integers specific operators (==#,*#,-#,+#) such that they put effects on integers we do see that we only produce constant bounds for the local and return regions. Our analysis does map the additional regions of both methods to unbounded which is optimal because these regions are used to store the thunks created within the factorial methods.

Figure 12.1: Factorials & Fibonacci

```
fib :: Int -> Int
fib n = if n == 0
        then 1
        else if n == 1
             then 1
             else fib (n-2) + fib (n-1)

fac :: Int -> Int
fac n = if n == 0
        then 1
        else n * fac (n-1)
```

12.2 Metrics

12.2.1 Dollar & id

The example derivation presented in Section 9.3 is matched by the implemented analysis. There is a small difference between '`id`' and '`($) id`', because `id` creates a thunk that must be stored in a region.

12.2.2 Impact on compilation speed

The compilation speed tests were run on a laptop with an i7-10750H clocked at 2.6GHz and 16GB of DDR4. The impact on compilation speed greatly depends on the value chosen for F as presented in Figure 12.2.

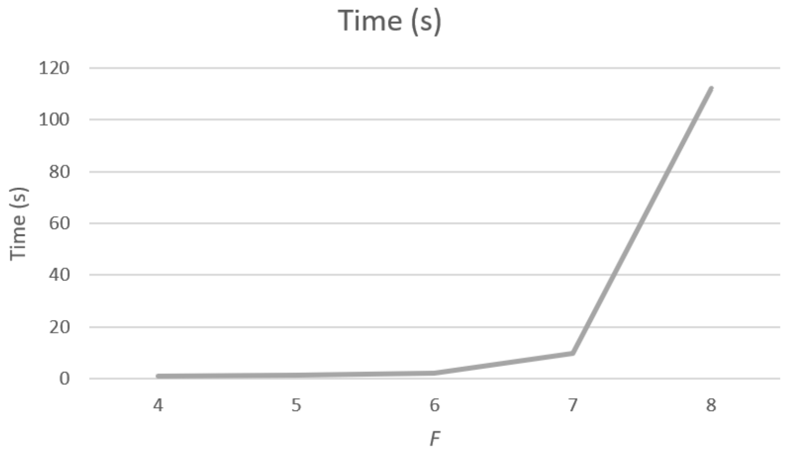

Figure 12.2: Compilation based on $K = 1$ and parameter F

The time seems to grow exponentially in F. For $F > 8$ the program crashes due to it running out of memory. For $F = 6$ Prelude compiles in two seconds on average. For $F = 7$ this goes up to ten seconds. The extra accuracy given by more iterations may still matter for user defined methods, however the extra impact on compilation time does not seem worth the low number of extra bounded regions. The exponential increase in compilation time has to do with some fixpoints growing exponentially in size.

Parameter K does not have any significant impact on compilation time. We have to do more fixpoint iterations for some methods before we reach a fixpoint, but if these fixpoints terminated for a lower value of K then these methods normalize properly, which means the annotations do not grow enough to cause a significant difference in compilation time. Increasing the number of fixpoint iterations could be connected to the optimization parameters of Helium.

12.2.3 Region bounds

Currently when compiling Prelude with $K = 1$ and $F = 6$ our analysis finds 2102 bounded local regions, of which 1133 have a bound of zero. Our analysis also finds 153 unbounded regions, either due to the bound exceeding K or failed fixpoint iteration. This means that we have 93% bounded regions. If we leave out the regions with a bound of zero we have 82% bounded regions.

Increasing K only results in fewer fixpoints being found. For $F = 7$ and $K = 2$ we find the same results. Increasing K results in more fixpoints failing and an increased number of unbounded regions. The drastic difference between $K = 5$ and $K = 6$ comes from fixpoint iteration failing for $+\!\!+$, which is used a lot throughout Prelude. The method $+\!\!+$ has many additional regions that are only used in the recursive calls to $+\!\!+$, which are correctly analyzed as zero for $K < 6$, but put to top for $K \geq 6$ because they are used in the body of that method. The number of bounded and unbounded regions does not change after $K = 7$, as the solution is still the same for $K = 10^9$.

Choosing a higher value for F is not reasonable due to the time it would take for analysis to analyze Prelude. Experimenting with $F = 8$ resulted in the same bounds. Note that the drastic change in bounded occurs when K gets to F. This is what we would expect, because if a single regions increases by one each iteration it will not reach K if $F < K$. We see the same effect when going from $K = 4$ to $K = 5$ with the parameter F set to 6. The reasons that this occurs at $K = F - 1$ is because in the first iteration $\perp$ is applied to the fixpoint. This means that we should pick a value for K such that $K \leq F - 2$.

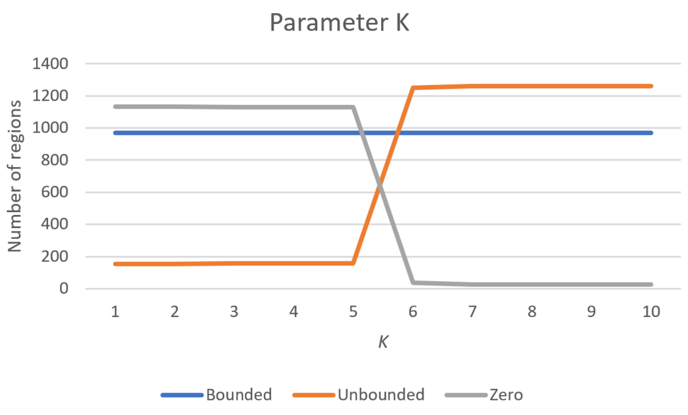

Figure 12.3: Number of regions based on $F = 7$ and parameter K

Region variables

If we also take into account the bounds on variables (addition regions, return regions but also other variables) the story changes. We derive a total of 9585 bounds of which only 3514 (36%) are bounded, however, this percentage is heavily skewed by a couple of factors: top annotations that put bounds on (non-region) annotation variables, not being able to analyze type class dictionaries and top annotations for primitive methods, e.g. `primPatternFailPacked` which is called by nearly all methods with a match instruction.

If we increase K there is a small increase in the number of bounded regions, this difference ($< 1\%$) however is not significant enough to risk other fixpoints failing. The drastic decrease in bounded regions when K exceeds $F - 2$ still occurs.

If we ignore the effects of type class dictionaries, the percentage of bounded regions could be up to 45%. If we also ignore the bounds on non-region annotation variables, the percentage of bounded could be up to 60%. It is unknown if these percentages are actually achievable, because it is unknown how many (un)bounded regions come from these two sources.

12.2.4 Fixpoints

Out of the 57 evaluated fixpoints 24 fixpoint iterations failed. Of the 24 failures 19 are higher order methods. The other methods are: `primAppend` from HeliumLang and `intercalate, (!!)` and `unwords` from Prelude. There is one more failed fixpoint which is an unnamed method called by the exponent operator.

It is hard to evaluate why a certain fixpoint does not terminate because the annotations of these methods are generally quite large. Some fixpoints of methods with higher order recursion fail to terminate because to compute the effect of such a fixpoint we must know the effect of the function argument f. This results in an endless string of applications, i.e. $\{\dots\} \sqcup (\pi_1[f\langle x\rangle\langle\rho\rangle] \oplus \{\dots\} \sqcup (\pi_1[f\langle x\rangle\langle\rho\rangle] \oplus \{\dots\} \sqcup \dots))$. Fixpoint iteration on such a fixpoint will never terminate because we do not have rules to evaluate the addition of annotations other than constraint sets.

A similar problem occurs with the join operator because projections are normalize outwards and applications outwards. This is the reason why the fixpoints for `intercalate, unwords` and `primAppend` fail. Normalizing outwards prevents joins on projections from being combined, which may also result in an exponential growth in annotation size if the fixpoint variable occurs more than once in this tuple. The exponential growth in annotation size may result in an exponential growth in evaluation time, which is the reason why the compilation time increases drastically when we increase F.

12.2.5 Datatypes

Simple datatypes without recursion are handled excellently by the analysis. They are not very interesting as they are no more than a tuple of their fields; they are evaluated like any ordinary tuple in the analysis.

Lists

The cons-constructor of the list datatype introduces a join between the new element and the old list. A list can be a parameter to a function, in which case that join becomes a join between a variable an tuple which cannot be evaluated. These kinds of joins did not seem to a be a problem for fixpoint iteration.

A more interesting case is where we append some function to a list of functions. When we append a function to nil, we take the join of that function with bottom. If the annotation of that function puts a bound of one on its return region, then that bound is worse than the implicit bound of zero from the bottom annotation, which means the join of bottom and our function puts a bound of one on the return region. Say we now append another function to the list, we then take the join of that function with the function that is already in the list. If the annotation of that new function maps the return region to unbounded, the return region in the resulting datatype tuple will be unbounded as well.

The result of this process is that the list always contains the worst bounds on the regions of all the elements of the lists, which makes sure that when we deconstruct a list we always project out an element that has bounds which are sound. Inspecting the annotations of functions that use lists shows that this works as expected in our analysis.

Type class dictionaries

There are no other recursive datatypes used in prelude besides type class dictionaries. Note that in all 36 cases where a top annotation is generated by datatypes it comes from type class dictionaries.

12.3 Reflection on implementation

Our implementation is capable of analyzing all Iridium methods. For any method that contains a datatype we cannot analyze we generate a top annotation locally instead of a top annotation for the entire method. The implementation does not contain any known bugs.

12.3.1 De Bruijn indices

De Bruijn indices were rather hard to get right. Many times when the implementation seemed stable we would come across another edge case.

Another source of issues coming from De Bruijn indices was the decision to use only one set of indices for both quantifications and abstractions. This made it much more difficult to derive an annotation from a type (i.e. top, bottom and fixpoint sorts), because the quantification must be offset by the number of abstraction between them, which is rather complicated and error sensitive. In other words: it is not possible to derive a sort directly from a type. Even though it was technically possible, we decided to switch to a separate set of indices for quantifications and abstractions.

12.3.2 Join of equal terms

During the evaluation of our implementation we discovered that we could achieve higher accuracy by adding the rule '$a \sqcup a = a$' to our evaluation rules. Early results show that with such a rule we evaluate three more fixpoints (`intercalate, unwords` and `primAppend`) and only have 124 unbounded regions (opposed to 153). The regions that became bounded were mapped to a bound of zero. There was no significant increase or decrease in compilation time.

13 Conclusion

Now that we have evaluated our results we can answer the research questions posed in Chapter 4.

Question 13.1. *Can region bound inference be made higher ranked?*

We have successfully created a higher ranked analysis that produces a high percentage of bounded regions. The analysis produces optimal results in the problematic cases presented by Veljstrup [22].

Vejlstrup only reports on the number of allocated (un)bounded regions instead of the number of (un)bounded regions. The number of allocated regions could be skewed towards bounded regions, because they are more likely to have a smaller lexical scope. Regions with a smaller lexical scope are more likely to be (de)allocated more often than those with a larger lexical scope. Part of the test set for Vejlstrups analyses was specifically written to work with RBMM. Due to these facts we cannot conclude if our analysis outperforms the work by Vejlstrup in general.

Question 13.2. *Does RBMM have a significant impact on compilation speed?*

The impact on compilation speed heavily depends on the chosen number of fixpoint iterations. If we choose $F = 6$ the impact is merely two seconds (15% of the total compilation time), while producing optimal results for Prelude. This choice of F is motivated by our evaluation results.

Question 13.3. *Does RBMM have a significant impact on program speed?*

Because code generation is not implemented for the region based memory management system this cannot be tested.

13.1 Discussion

We have presented a novel approach to region bound inference. From our evaluation results we can see that the analysis performs well, but there is still room for improvement. We will discuss a number of possible improvements in our future work in Section 14. When compared to previous work our analysis performs admirably, with the added bonus of separate compilation.

The speed of the analysis is also satisfactory, but it can still be improved. The exponential increase in compilation time when F is increased comes from fixpoints that cause an exponential increase in annotation size. There is still room for code optimization, especially in the annotation evaluation code. Solving these issues might allow us to do more fixpoint iterations to solve more complex fixpoints.

From analysis results we can see that Vejlstrups comment about choosing $K = 1$ still holds, however, due to different reasons in our analysis. A low value for K ensures that more fixpoints will terminate, a higher value for K increases the risk of a fixpoint failing, which in general has worse effects than setting a single bound to unbounded. The value of K should not exceed $F - 2$ if a higher value for K is desired.

For $K = 1$ we do not need to convert the number of allocations per regions to a number of allocated bytes. It may however still be desirable to extend the analysis such that it also returns the number of bytes required for each region. The program transformation can then convert bounded regions that are larger then some fixed number of bytes to unbounded, which would prevent the stack from being filled by unreasonably large regions.

There is still room for improvement when it comes to precision. Termination of fixpoint iteration in essential for good results, the top annotation is often a large overestimation. Adding support for simple recursive datatypes is a nice to have, but did not impact or results because the list is the only recursive datatype in the Helium Prelude.

14 Future work

14.1 Code generation

The final step to complete a working region based memory management system is to implement the code generation. The code generation should use the information provided by the region and region bound analyses. The implementation of this system must work with LLVM.

Objects in the global region should also be dealt with properly. These objects will not be deallocated before the program terminates, which in many cases will be a large overestimation. To combat this problem these region could be handled by traditional (generational) garbage collection.

Another problem is region waste, which is unused space in region pages of heap allocated regions. Region waste can as high as 25% of the total memory usage in some cases[4]. A possible solution is to replace all unbounded regions with the global region. The objects in this global region could then be managed by GC. This however does mean GC has to manage more objects, which could negatively impact execution time. One could also to reduce the allocation size of region pages which may increase the number of times region pages are allocated. Another solution could be to run a process similar to GC that deallocates the unused space within region pages, which may cause more memory fragmentation.

14.2 Non-static bounds

To deal with unbounded regions one could devise a analysis that would improve the handling of such regions. Similar analyses are discussed in Section 3.4 of the related work. Such analyses could be put in the Iridium pipeline after the region bound analysis, such that it can ignore any bounded regions discovered by that analysis.

14.3 Precision

There are still many ways to further increase the strength of the analysis. In this section we will highlight a couple ways.

14.3.1 Addition rules

To further strengthen evaluation one could introduce extra rules for addition. These rules could allow for addition of annotations with a different sort than C. Our analysis only derives additions with sort C, however the annotation is not always a constraint set. The rules would allow to break apart parts of an annotation that cannot be broken apart now, for example $\psi\langle\rho\rangle \oplus \psi\langle\rho\rangle$, where $\psi\langle\rho\rangle$ is of sort C.

Breaking apart these annotations would likely require the introduction of a multiplication operator to allow for addition of variables. This annotation could for instance be evaluated to $(2 \times \psi)\langle\rho\rangle$, where '$\times$' is the multiplication operator. A multiplication $n \times a$ with $n > K$ can be evaluated to $\infty \times a$, because any constraint with a finite bound multiplied by n will result in a bound larger than K. This multiplication rule would allow us to analyze recursive higher order methods, because it combats the issue discussed in the evaluation chapter.

14.3.2 Bound on global region

If we are able to derive a bound on the global region we can optimize allocations even further. Normally, each allocation has to check if there is enough space on the heap to allocate the corresponding object. If we are able to derive a bound on the global region we only have to check if there is enough space on the heap when some function is called.

Another possibility is that a program can be entirely stack allocated, but this is only realistic for smaller programs. The region bound analysis does already produce the required information, it only has to be propagated through the modules.

14.3.3 Datatypes

Currently, the analysis only supports simple non recursive datatypes and list. It would be good to add support for simple recursive datatypes, because region inference supports this class of datatypes too.

Type class dictionaries

As discussed in Section 7.3.3, it is very difficult to analyze datatypes with recursion in the contravariant position of a function field. Currently, all type class dictionaries are desugared to be recursive in such a manner. The 'Eq'-type class dictionary would be desugared as seen in Example 14.1.

Figure 14.1: Type class disctionaries

```
// Eq definition
class  Eq a  where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

// Desugares to
DictEq a = DictEq (DictEq a -> a -> a -> Bool)
                  (DictEq a -> a -> a -> Bool)
```

There are two solutions to this problem: inlining and removing the dictionary argument from the fields. Inlining only helps if we know the type of the dictionaries, we can the inline the method and not use the type class dictionary. This however, only helps if we know the type annotations.

The other method requires a change in the way dictionaries are desugared. It is possible to get rid of the dictionary argument, which would desugared all type class dictionaries into simple non-recursive datatypes. We know how to handle these kinds of datatypes.

14.3.4 Hand written annotations

There are primitive method calls (e.g. to LLVM methods) that cannot be analyzed. For optimal region bound inference these annotations should be created by hand. This could be done for popular methods from Prelude as well for which the analysis is unable to derive the optimal result.

14.3.5 Higher order fixpoints

Currently fixpoints are calculated immediately when analyzing a recursive method. For higher order functions this means that fixpoint iteration will not terminate. Some of these higher order functions suffer for the lack of evaluation rules for addition, while others do not have enough information to evaluate the fixpoint (e.g. `foldl`). The only solution is top.

If such a higher order method does not contain any local region we can choose to not calculate the fixpoint immediately; the method does not need to know the effects. For such a method we store an annotation that contains a fixpoint construct. If this method is used within some other method the function argument may be applied, which could mean that we have enough information to compute the fixpoint.

Delaying fixpoint iteration does not come for free. If a such a higher order recursive method is used more than once, then the fixpoint will be evaluated more than once. There are no guarantees that enough information is supplied to the fixpoint to terminate fixpoint iteration, which in turn means that we only see an increase in compilation time without any gain in accuracy.

14.4 Optimalisation

There are still many ways we can optimize the current implementation. A good example of a possible optimization is the annotation evaluation code. Currently the evaluation rules are computed by a fold over the annotation structure. This creates quite minimal code and ensures all sub-annotations of some annotation are evaluated before an annotation is evaluated. This however means we are also giving away fine grained control over the evaluation rules. More efficient handling for example could be done for projections, where we can avoid evaluating sub-annotations of the tuple that are not needed.

Bibliography

[1] J. S. Miller and G. J. Rozas. "Garbage Collection is Fast, But a Stack is Faster". In: *A.I. Memo No. 1462* (1994).

[2] A. W. Appel. *Simple generational garbage collection and fast allocation.* Mar. 1988. DOI: 10.1002/spe.4380190206. URL: https://doi.org/10.1002/spe.4380190206.

[3] P. M. Sansom and S. L. Peyton Jones. "Generational Garbage Collection for Haskell". In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture.* FPCA '93. Copenhagen, Denmark: Association for Computing Machinery, 1993, 106–116. ISBN: 089791595X. DOI: 10.1145/165180.165195. URL: https://doi.org/10.1145/165180.165195.

[4] M. Tofte et al. "A Retrospective on Region-Based Memory Management". In: *Higher-Order and Symbolic Computation* 17.3 (Sept. 2004), pp. 245–265. ISSN: 1573-0557. DOI: 10.1023/B:LISP.0000029446.78563.a4. URL: https://doi.org/10.1023/B:LISP.0000029446.78563.a4.

[5] B. Davey and H. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 2002. ISBN: 9781107717527. URL: https://books.google.co.uk/books?id=BueMAgAAQBAJ.

[6] F. Nielson and H. R. Nielson. "Type and Effect Systems". In: *Correct System Design: Recent Insights and Advances.* Ed. by E.-R. Olderog and B. Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 114–136. ISBN: 978-3-540-48092-1. DOI: 10.1007/3-540-48092-7_6. URL: https://doi.org/10.1007/3-540-48092-7_6.

[7] S. P. Jones et al. "Practical type inference for arbitrary-rank types". In: *Journal of Functional Programming* 17.1 (2007), 1–82. DOI: 10.1017/S0956796806006034.

[8] S. Holdermans and J. Hage. "Polyvariant Flow Analysis with Higher-Ranked Polymorphic Types and Higher-Order Effect Operators". In: *SIGPLAN Not.* 45.9 (Sept. 2010), 63–74. ISSN: 0362-1340. DOI: 10.1145/1932681.1863554. URL: https://doi.org/10.1145/1932681.1863554.

[9] R. Koot. *Higher-ranked exception types.* "*Available online:* https://github.com/ruudkoot/phd". 2015.

[10] F. Thorand and J. Hage. "Higher-Ranked Annotation Polymorphic Dependency Analysis". In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings.* Ed. by P. Müller. Vol. 12075. Lecture Notes in Computer Science. Springer, 2020, pp. 656–683. DOI: 10.1007/978-3-030-44914-8_24. URL: https://doi.org/10.1007/978-3-030-44914-8_24.

[11] I. G. de Wolff. "Higher ranked region inference for compile-time garbage collection". MA thesis. Dept. of Computer Science, University of Utrecht, 2019.

[12] D. T. Ross. "The AED Free Storage Package". In: *Commun. ACM* 10.8 (Aug. 1967), 481–492. ISSN: 0001-0782. DOI: 10.1145/363534.363546. URL: https://doi.org/10.1145/363534.363546.

[13] D. Grossman et al. "Region-Based Memory Management in Cyclone". In: *SIGPLAN Not.* 37.5 (May 2002), 282–293. ISSN: 0362-1340. DOI: 10.1145/543552.512563. URL: https://doi.org/10.1145/543552.512563.

[14] D. Gay and A. Aiken. "Language Support for Regions". In: *SIGPLAN Not.* 36.5 (May 2001), 70–80. ISSN: 0362-1340. DOI: 10.1145/381694.378815. URL: https://doi.org/10.1145/381694.378815.

[15] S. Kowshik, D. Dhurjati, and V. Adve. "Ensuring Code Safety without Runtime Checks for Real-Time Control Systems". In: *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems.* CASES '02. Grenoble, France: Association for Computing Machinery, 2002, 288–297. ISBN: 1581135750. DOI: 10.1145/581630.581678. URL: https://doi.org/10.1145/581630.581678.

[16] H. Makholm. "Region-Based Memory Management in Prolog". MA thesis. Dept. of Computer Science, University of Copenhagen, 2000.

[17] Q. Phan, Z. Somogyi, and G. Janssens. "Runtime Support for Region-Based Memory Management in Mercury". In: *Proceedings of the 7th International Symposium on Memory Management.* ISMM '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, 61–70. ISBN: 9781605581347. DOI: 10.1145/1375634.1375644. URL: https://doi.org/10.1145/1375634.1375644.

[18] W. S. Beebee and M. Rinard. "An Implementation of Scoped Memory for Real-Time Java". In: *Embedded Software.* Ed. by T. A. Henzinger and C. M. Kirsch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 289–305. ISBN: 978-3-540-45449-6.

[19] L. Birkedal and N. Rothwell. *The ML Kit - Version 1.* http://elsman.com/mlkit/. 1993.

[20] M. Elsman and N. Hallenberg. "On the Effects of Integrating Region-Based Memory Management and Generational Garbage Collection in ML". In: *Practical Aspects of Declarative Languages.* Ed. by E. Komendantskaya and Y. A. Liu. Cham: Springer International Publishing, 2020, pp. 95–112. ISBN: 978-3-030-39197-3.

[21] N. Hallenberg. *A Region Profiler for a Standard ML compiler based on Region Inference.* https://elsman.com/mlkit/pdf/profiling.pdf. 1996.

[22] M. Vejlstrup. "Multiplicity Inference - Inferring size of regions". MA thesis. University of Copenhagen, 1994.

[23] M. Tofte et al. *Programming with Regions in the ML Kit (for Version 3).* Tech. rep. Technical Report 98/25. Department of Computer Science, University of Copenhagen, Dec. 1998.

[24] I. G. de Wolff. *The Helium Haskell compiler and its new LLVM backend. NL 2019 EuroLLVM Developers' Meeting. Available online:* https://youtu.be/x6CBks1paF8.

[25] L. Birkedal, M. Tofte, and M. 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.* POPL '96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, 171–183. ISBN: 0897917693. DOI: 10.1145/237721.237771. URL: https://doi.org/10.1145/237721.237771.

[26] K. Crary, D. Walker, and G. Morrisett. “Typed Memory Management in a Calculus of Capabilities”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, 262–275. ISBN: 1581130953. DOI: `10.1145/292540.292564`. URL: `https://doi.org/10.1145/292540.292564`.

[27] A. Aiken, M. Fähndrich, and R. Levien. “Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages”. In: *SIGPLAN Not.* 30.6 (June 1995), 174–185. ISSN: 0362-1340. DOI: `10.1145/223428.207137`. URL: `https://doi.org/10.1145/223428.207137`.

[28] F. Henglein, H. Makholm, and H. 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.* PPDP ’01. Florence, Italy: Association for Computing Machinery, 2001, 175–186. ISBN: 158113388X. DOI: `10.1145/773184.773203`. URL: `https://doi.org/10.1145/773184.773203`.

[29] S. Jost et al. “Static Determination of Quantitative Resource Usage for Higher-Order Programs”. In: *SIGPLAN Not.* 45.1 (Jan. 2010), 223–236. ISSN: 0362-1340. DOI: `10.1145/1707801.1706327`. URL: `https://doi.org/10.1145/1707801.1706327`.

[30] J. Hoffmann and M. Hofmann. “Amortized Resource Analysis with Polynomial Potential”. In: *Programming Languages and Systems.* Ed. by A. D. Gordon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 287–306. ISBN: 978-3-642-11957-6.

[31] J. Hoffmann, K. Aehlig, and M. Hofmann. “Multivariate amortized resource analysis”. In: *ACM Trans. Program. Lang. Syst.* 34 (2012), 14:1–14:62.

[32] J.-Y. Girard. “The system F of variable types, fifteen years later”. In: *Theoretical Computer Science* 45 (1986), pp. 159–192. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/0304-3975(86)90044-7`. URL: `https://www.sciencedirect.com/science/article/pii/0304397586900447`.

[33] R. Hinze and R. Paterson. “Finger trees: a simple general-purpose data structure”. In: *Journal of Functional Programming* 16.2 (2006), 197–217. DOI: `10.1017/S0956796805005769`.

[34] H. Ottens. *region-size branch, Helium on GitHub. Available online:* `https://github.com/Helium4Haskell/helium/tree/region-size`.

[35] B. Heeren, D. Leijen, and A. v. IJzendoorn. “Helium, for Learning Haskell”. In: *Proceedings of the ACM SIGPLAN Haskell Workshop (Haskell’03), Uppsala, Sweden.* ACM SIGPLAN, Aug. 2003, p. 62. URL: `https://www.microsoft.com/en-us/research/publication/helium-for-learning-haskell/`.

[36] *The Helium Prelude. Available online:* `https://github.com/Helium4Haskell/helium/tree/region-size/lib`.

[37] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04).* Palo Alto, California, Mar. 2004.

[38] *LLVM’s Analysis and Transform Passes. Available online:* `https://llvm.org/docs/Passes.html`.

A

Syntactic substitution of annotations

$$\frac{}{\psi[\psi := a] \longrightarrow a}$$

$$\frac{}{\phi[\psi := a] \longrightarrow \phi}$$

$$\frac{}{\rho[\psi := a] \longrightarrow \rho}$$

$$\frac{}{(\Lambda\psi : s.\ a)[\psi := a'] \longrightarrow \Lambda\psi : s.\ (a[\psi := a'])}$$

$$\frac{}{a_1\langle a_2\rangle[\psi := a'] \longrightarrow a_1[\psi := a']\langle a_2[\psi := a']\rangle}$$

$$\frac{}{()[\psi := a] \longrightarrow ()}$$

$$\frac{}{(a_1, \ldots, a_n)[\psi := a] \longrightarrow (a_1[\psi := a], \ldots, a_n[\psi := a])}$$

$$\frac{}{\pi_i(a)[\psi := a'] \longrightarrow \pi_i(a[\psi := a'])}$$

$$\frac{}{(a_1 \oplus a_2)[\psi := a'] \longrightarrow a_1[\psi := a'] \oplus a_2[\psi := a']}$$

$$\frac{}{(a \setminus \rho)[\psi := a'] \longrightarrow a[\psi := a'] \setminus \rho}$$

$$\frac{}{(a_1 \sqcup a_2)[\psi := a'] \longrightarrow a_1[\psi := a'] \sqcup a_2[\psi := a']}$$

$$\frac{}{(\forall\alpha.a)[\psi := a'] \longrightarrow \forall\alpha.(a[\psi := a'])}$$

$$\frac{}{(a\ \{\tau\})[\psi := a'] \longrightarrow a[\psi := a']\ \{\tau\}}$$

$$\frac{}{\top[C : s][\psi := a'] \longrightarrow \top[c[\psi := a'] : s]}$$

$$\frac{}{\bot[s][\psi := a'] \longrightarrow \bot[s]}$$

$$\frac{}{(fix\ s.\ a)[\psi := a'] \longrightarrow fix\ s.\ (a[\psi := a'])}$$

A.1 Constraint sets

$$\frac{}{(C \cup \{\psi \mapsto n\})[\psi := \rho] \longrightarrow C[\psi := \rho] \oplus \{\psi \mapsto n\}[\psi := \rho]}$$

$$\frac{\psi \notin C}{C[\psi := \rho] \longrightarrow C}$$

$$\frac{}{\{\psi \mapsto n\}[\psi := \rho] \longrightarrow \{\rho \mapsto n\}}$$

$$\frac{1 \leq k < n}{\{\psi.k \mapsto n\}[\psi := (\hat{\rho_1}, \ldots, \hat{\rho_n})] \longrightarrow \{\psi \mapsto n\}[\psi := \hat{\rho_k}]}$$

$$\frac{}{\{\psi \mapsto n\}[\psi := (\hat{\rho_1}, \ldots, \hat{\rho_n})] \longrightarrow \mathit{collect}((\hat{\rho_1}, \ldots, \hat{\rho_n}), n)}$$

$$\frac{}{\{\psi \mapsto n\}[\psi := a] \longrightarrow \{\psi \mapsto n\}[\psi := \mathit{gatherConstraintsTuple}(a)]}$$

$$\begin{aligned} \mathit{collect}(\rho, n) &= \{\rho \mapsto n\} \\ \mathit{collect}((\rho_1, \ldots, \rho_n), n) &= \mathit{collect}(\rho_1, n) \oplus \cdots \oplus \mathit{collect}(\rho_n, n) \end{aligned} \tag{A.1}$$

B

Dollar derivation

The derivation of dollar can be found on the next page.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f \mapsto \psi_f \in \Delta_2}{\Delta_2 \vdash f \hookrightarrow \psi_f \ \&\ \bot}\,\textsc{var} \qquad \dfrac{x \mapsto \psi_x \in \Delta_2}{\Delta_2 \vdash x \hookrightarrow \psi_x \ \&\ \bot}\,\textsc{var} \qquad (a_r, \gamma_r) = \psi_f\langle\psi_x\rangle\langle\phi_x\rangle \qquad \hat{\rho}_2 \mapsto \phi_x \in \mathcal{P}}{\Delta_2 \vdash f[\hat{\rho}_2]\ x \hookrightarrow a_r \ \&\ \bot \oplus \bot \oplus \gamma_r \qquad \mathit{returntype} = \beta}\,\textsc{apl}}{\Delta_1 \vdash \lambda[\hat{\rho}_2]x.\ f[\hat{\rho}_2]\ x \hookrightarrow \Lambda\psi_x : \Phi_\Gamma(\alpha).\Lambda\phi_x : P_\Gamma(\beta).(a_r, \gamma_r) \ \&\ \bot \qquad \mathit{returntype} = \alpha \to \beta}\,\textsc{abs}}{\{\};\{\} \vdash \lambda[\hat{\rho}_1]f.\lambda[\hat{\rho}_2]x.\ f[\hat{\rho}_2]\ x \hookrightarrow \Lambda\psi_f : \Phi_\Gamma(\alpha \to \beta).\Lambda\phi_f : P_\Gamma(\alpha \to \beta).(\Lambda\psi_x : \Phi_\Gamma(\alpha).\Lambda\phi_x : P_\Gamma(\beta).(a_r, \gamma_r), \bot) \ \&\ \bot}\,\textsc{abs}}{\{\};\{\} \vdash \forall\beta.\lambda[\hat{\rho}_1]f.\lambda[\hat{\rho}_2]x.\ f[\hat{\rho}_2]\ x \hookrightarrow \forall\beta.\Lambda\psi_f : \Phi_\Gamma(\alpha \to \beta).\Lambda\phi_f : P_\Gamma(\alpha \to \beta).(\Lambda\psi_x : \Phi_\Gamma(\alpha).\Lambda\phi_x : P_\Gamma(\beta).(a_r, \gamma_r), \bot) \ \&\ \bot}\,\textsc{qnt}}{\{\};\{\} \vdash \forall\alpha.\forall\beta.\lambda[\hat{\rho}_1]f.\lambda[\hat{\rho}_2]x.\ f[\hat{\rho}_2]\ x \hookrightarrow \forall\alpha.\forall\beta.\Lambda\psi_f : \Phi_\Gamma(\alpha \to \beta).\Lambda\phi_f : P_\Gamma(\alpha \to \beta).(\Lambda\psi_x : \Phi_\Gamma(\alpha).\Lambda\phi_x : P_\Gamma(\beta).(a_r, \gamma_r), \bot) \ \&\ \bot}\,\textsc{qnt}
$$

$$
\Delta_1 = \{f \mapsto \psi_f\};\{\rho_1 \mapsto \phi_f\} \qquad \Delta_2 = \{f \mapsto \psi_f, x \mapsto \psi_x\};\{\rho_1 \mapsto \phi_f, \rho_2 \mapsto \phi_x\}
$$