Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

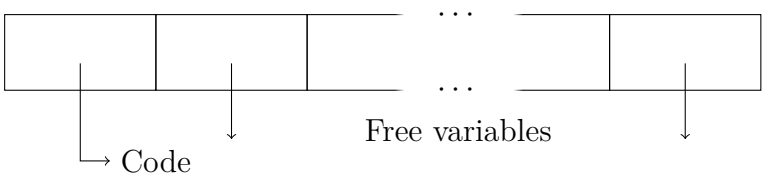
Lehrstuhl Prof. Dr.-Ing. Snelting

Escape Analysis for the Glasgow Haskell Compiler

Master-Arbeit von

Sebastian Scheper

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	M. Sc. Sebastian Graf
Abgabedatum:	23. Mai 2022

Abstract

Speicherverwaltung ist eine der anspruchsvollsten Bestandteile der Programmierung. Daher wurde die Zuständigkeit für Speicherverwaltung vor allem mit dem Aufkommen von Garbage-Collectors mehr auf die Laufzeitumgebungen verlagert. Besonders funktionale Programmiersprachen wie Haskell legen viel Wert auf die Abstraktion von Speicherverwaltung. Allerdings lässt der *Glasgow Haskell Compiler* zurzeit noch viel Raum für Optimierung von Speicherverbrauch und Performanz.

Zur Ermöglichung solcher Optimierungen befasst sich diese Master-Thesis mit der Entwicklung einer statischen Escape-Analyse für den Glasgow Haskell Compiler. Der Hauptzweck dieser Analyse soll die Identifikation von Speicherallokation auf dem Heap sein, die stattdessen auf dem Stack getätigt werden können.

Auswertungen mithilfe eines instrumentierten Interpreters zeigen, dass dank der Analyse durchschnittlich 13,7 % des Heap-allokierten Speichers auf den Stack verlagert werden kann.

Memory management is one of the most challenging parts of programming. This is why the responsibility of memory management is being delegated to the runtime environments, mainly due to the rise of garbage collectors. Especially functional programming languages like Haskell are emphasizing the abstraction of memory management. The *Glasgow Haskell Compiler*, however, leaves a lot of room for optimization with regard to memory usage and performance.

To enable such optimization, this master thesis deals with the development of a static escape analysis for the Glasgow Haskell Compiler. The main purpose of this analysis shall be the identification of memory allocations on the heap, that can be performed on the stack instead.

The evaluation via an instrumented interpreter show that due to the analysis, on average 13.7% of heap allocated memory can be shifted to the stack.

Contents

1. Introduction

Consider this Haskell code:

```
f p (x:xs)  =   let px = p x
                in  case any px xs of   True    -> Just x
                                        _       -> f p xs
f p []      =   Nothing
```

The body of the function `f` consists of a let expression, that defines a variable `px`. When the function is being called, first a closure gets created, which represents `px` at runtime. After that, the let body gets evaluated. Currently, to create a closure, the GHC will generate a heap allocation. However, by taking a closer look at the let body, it becomes apparent that `px`'s closure can only be evaluated during the evaluation of the let expression. The access to `px`'s closure is limited to the lexical scope of `px`. Hence, a stack allocation would be sufficient.

In a different situation, `f` may grant its caller access to `px`'s closure by returning a value, that contains a reference to `px`'s closure:

```
f p (x:xs)  =   let px = p x
                in  Just px
f p []      =   Nothing
```

Here, said closure can possibly be accessed outside `px`'s lexical scope.

The latter case is the exact reason why the GHC, among other compilers for different languages, would generate a heap allocation for `px`. However, in the former case, a stack allocation would be a valid optimization to the code, it would be much cheaper than a heap allocation. The purpose of an escape analysis is to basically distinguish these two cases, to identify bindings, that can be allocated on the stack. To our knowledge, there is no escape analysis for a Haskell compiler or any compiler for a lazy language yet.

This thesis makes the following contributions:

- We contemplate the benefits an escape analysis might have for a Haskell Compiler (Section 2.2). We also give an understanding of the challenges to overcome for the development of an escape analysis in the context of Haskell (Section 2.5). The most important of them are the identification of memory allocations in the code, being aware of the lexical scope, how to treat closures and thunks and how the semantics of *join points* affect the stack.

- We present the design and specification for an escape analysis algorithm, that processes STG code to identify bindings, that can be allocated on the stack

(Section 3.1). To our knowledge, this is the first escape analysis to be developed for a lazy language like Haskell. We introduce core concepts of the algorithm and discuss the design decisions that aid overcoming the challenges introduced previously (Section 3.2).

- To evaluate the soundness and effectiveness of the algorithm, we contemplate the significance of different metrics (Section 4.1). To measure these metrics, we introduce a custom profiling feature for an STG interpreter (Section 4.2). The general approach is to keep track of the access to heap objects at runtime in consideration of the lexical scope of their binding. We present the measurements of these metrics for a variety of benchmark programs (Section 4.4).

2. Basics and related work

First, we cover the basic concepts of escape analysis Then in section 2.2, we discuss how a compiler can benefit from it. Section 2.3 gives a quick explanation of different notations that are used in the remainder of this thesis. Section 2.4 provides an overview of STG language. Lastly, section 2.5 describes the challenges to overcome when performing an escape analysis.

2.1. Escape analysis

Escape analysis can be seen as a specialization of points-to analysis, in the way that we statically determine for a program, which data items can be pointed to by a specific pointer. [1] defines escape analysis as a static analysis that determines, whether the lifetime of data may exceed its static scope. So basically, we aren't interested in the concrete points-to set of a pointer, the only relevant information is, whether a data item is contained in the points-to set of a pointer, that is located outside the lexical scope of said data item or not. This relationship between escape analysis and points-to analysis is demonstrated using an example in Java:

```
private static Integer create1() {
    Integer p1 = new Integer(1); // o1
    System.out.println("Created␣" + p1);
    return 1;
}

private static Integer create2() {
    Integer p2 = new Integer(2); // o2
    System.out.println("Created␣" + p2);
    return p2;
}

public static void main(String[] args) {
    Integer p3;

    switch (args[1]) {
        case "1":
            p3 = createInteger1();
            break;
        case "2":
            p3 = createInteger2();
```

```
                break;
        }
}
```

There are data items created in two different places in the program, which are the objects `o1` and `o2`. After performing a points-to analysis, the points-to sets for the variables `p1`, `p2` and `p3` are:

$$PT(\texttt{p1}) = \{\texttt{o1}\}, \quad PT(\texttt{p2}) = \{\texttt{o2}\}, \quad PT(\texttt{p3}) = \{\texttt{o2}\}$$

Also note, that `p3` is defined outside the lexical scope of `p1` and `p2`. From the fact, that `o2` is in $PT(\texttt{p3})$ and `o1` isn't, we can derive that `o2` escapes and `o1` doesn't.

All an escape analysis effectively does, is assigning the property *escapes* or *doesn't escape* to an object's point of creation in the program code. (e.g. call to `malloc` in C or the `new` operator in most OO languages). However, identifying memory allocations for non-imperative languages like Haskell, is not as straight forward as just looking for certain operators in the code. This problem will be discussed in section 2.5.1

2.2. Possible optimizations

For the code produced by the GHC there are two major optimizations, that can be done based on escape analysis under certain circumstances:

Stack allocation Non-escaping objects can be allocated on the Haskell RTS stack instead of the managed heap.

Reuse of surrounding closures Non-escaping closures don't have to capture their free variables on their own. For variables that are already captured by the surrounding closure, they can refer to their surrounding closure instead. This concept is called *linked closure*.

In imperative languages there might also be the opportunity for the elimination of synchronization [2], which is less relevant for Haskell, since Haskell programmers are expected to work with immutable data.

There are some more or less obvious benefits to be achieved by performing these optimizations: The decrease of allocations on the managed heap takes load off the garbage collection and therefore reduces performance and memory overhead that usually comes with GC. Haskell uses a generational copying collector, which [3] describes in more detail.

A more subtle but not necessarily less relevant advantage of stack allocation in contrast to heap allocation is the increase of data locality which, according to [4], can lead to performance improvements.

The reuse of surrounding closures might decrease the memory usage: Usually, a closure captures its free variables in form of pointers or values. However, a non-escaping closure can only be entered in the context it has been defined in, the binders

of the free variables are in scope for at least the entire lifetime of the closure. Hence, there is no need to capture any of the free variables.

To implement these optimizations, the following conditions must hold for non-escaping objects:

- To safely allocate an object on the stack, it has to be determined at compile time to only be used as long as it is present on the stack.
- For a closure to reuse its surrounding closure instead of capturing its own free variables, it can only be entered while the surrounding closure is alive. Otherwise, the use of linked closures might keep the surrounding closure alive for longer than necessary, which would be a threat to the *space safety* [5].

The escape analysis developed as part of this thesis has many similarities with the escape analysis for the Caml Special Light compiler [6]. It operates on an abstract syntax tree which is based on lambda calculus. In addition to that, the analysed code is being optimized to perform stack allocations where possible. Comparing the execution of several benchmarks shows a significant speedup and decrease in memory usage for the optimized code.

2.3. Notations

Overline: A sequence of equal expressions with arbitrary length $e_1\ e_2 \ldots$ can be displayed with *overline* syntax as $\overline{e}$.

This way, a recursive let expression with a sequence of several bindings can be displayed in a more compact way:

$$\textbf{let rec}\ b_1 = r_1\ b_2 = r_2\ \ldots\ \textbf{in}\ e \quad \equiv \quad \textbf{let rec}\ \overline{b = r}\ \textbf{in}\ e$$

To write case expressions, nested overlines are necessary:

$$\begin{array}{lll} \textbf{case}\ s\ \textbf{of} & K_1\ a_{1,1}\ a_{1,2} \cdots \to e_1 & \\ & K_2\ a_{2,1}\ a_{2,2} \cdots \to e_2 & \equiv \quad \textbf{case}\ s\ \textbf{of}\ \overline{K\ \overline{a} \to e} \\ & \ldots & \end{array}$$

The exact number of elements in the sequence can be specified in the superscript of the overline. A range can be specified by stating an index:

$$\overline{a_j}^{n \le j \le N} \equiv a_n\ a_{n+1} \ldots a_N$$

The index also makes explicit, where the subscript should be attached:

$$\overline{f(a_j)}^{1 \le j} \equiv f(a_1)\ f(a_2) \ldots$$

Note, that $\overline{f(a)}$ couldn't distinguish this from $f_1(a_1)\ f_2(a_2) \ldots$

Representation of partial functions: A function f of the type $A \to B$ can be interpreted as a subset of $A \times B$ with

$$f(a) = b \leftrightarrow (a, b) \in f.$$

The difference of partial functions is that they aren't left-total. Wherever a partial function $f \in A \rightharpoonup B$ is not defined, it returns $\bot$ (the *bottom* element):

$$f(a) = \bot \leftrightarrow \nexists b \in B : (a, b) \in f$$

Function updates: For an arbitrary function or partial function f, an updated version can be created by putting the updated mapping in square brackets:

$$f[a \mapsto b](x) \equiv \begin{cases} b & \text{if } x = a \\ f(x) & \text{else} \end{cases}$$

$f[a \mapsto b]$ is a function that behaves like f, except for returning b when a is being passed to it.

2.4. The STG language

When designing a compiler for a non-strict functional language, a big challenge is bridging the gap between the functional and the imperative world. An abstract machine model can be helpful, because it abstracts a lot of details about the source language and the code generation. For this purpose, the GHC follows the model of the *Spineless Tagless G-machine* or short *STG-machine*. This machine model is special in the way, that it doesn't execute a sequence of imperative machine instructions, the STG-machine's programming language itself is a non-strict functional language. This language is called the *STG language*. [7] introduces both the denotational semantics and the operational semantics of the STG language based on the STG-machine. It also discusses, how to map the STG language onto stock hardware.

The Haskell compiler runs through the following stages:

1. Transforming the source language into a smaller *Core language* by eliminating Haskell's syntactic sugar, performing type checks, resolving overloading and translating pattern matching to case expressions
2. Applying several program analyses to the Core language
3. Translating the Core language to the *STG-Language*
4. Compiling the STG code to an imperative, portable assembly language such as `C--`
5. Assembling the code to machine instructions

Constructors	$K \in \mathsf{Con}$		
Variables	$a, b, f, x \in \mathsf{Var}$		
Literals	$l \in \mathsf{Lit}$		
Primitive Operations	$p \in \mathsf{Prim}$		
Right-hand sides	$r \in \mathsf{Val}$	$::=$	$\overline{x}\,\lambda\,\pi\,\overline{a}\,.\,e \mid K\ \overline{a}$
Update flags	$\pi \in \mathsf{Upd}$	$::=$	$\mathtt{u} \mid \mathtt{n}$
Expressions	$e, s \in \mathsf{Exp}$	$::=$	$\textbf{let}\ \overline{b = r}\ \textbf{in}\ e$
		$\mid$	$\textbf{let rec}\ \overline{b = r}\ \textbf{in}\ e$
		$\mid$	$\textbf{join}\ \overline{b = r}\ \textbf{in}\ e$
		$\mid$	$\textbf{join rec}\ \overline{b = r}\ \textbf{in}\ e$
		$\mid$	$f\ \overline{a}$
		$\mid$	$p\ \overline{a}$
		$\mid$	$\textbf{case}\ s\ \textbf{of}\ \overline{K\ \overline{a} \to e}$
Programs	$P \in \mathsf{Prog}$	$::=$	$\overline{b = r}$

Figure 2.1.: Syntax of the STG language

The following sections give a brief introduction to the syntax and semantics of the STG language.

The Syntax is shown in figure 2.1. Like lots of other intermediate representations for functional programming languages, STG can be described as *enriched lambda calculus*. It is a superset of the lambda calculus [8], which is still very minimal. Besides the common features of pure functional languages, STG introduces the following features:

- Primitive operations allow for performance critical functionality to be implemented in a more low-level imperative manner.
- Lambda expressions have the form $\overline{x}\,\lambda\,\pi\,\overline{a}\,.\,e$. They explicitly mention their free variables $\overline{x}$ and their *update flag* π. Those are relevant for the operational semantics of allocating and evaluating a lambda expression at runtime. A lambda expression in combination with this additional information is called a *closure*.
- It is made explicit whether a binding or a sequence of bindings is recursive.
- STG also distinguishes between regular let bindings and *join points*. From a denotational standpoint there is no difference between them, but operationally they allow for powerful optimization [9].
- A program is represented simply as a sequence of bindings.

By transforming a program from the Core language to the STG language, further changes are being made to the code:

- Each argument that is passed to a function or constructor must be an *atom* (either a variable or a literal). Since arguments in a Core program can be arbitrary expressions, new let bindings must be introduced for each non-trivial argument.

- All constructor calls and primitive operation must be saturated. Unsaturated constructors and primitive operations must be wrapped in a lambda expression by performing η-abstractions on them.

- Pattern matching is transformed into case expressions. Each case expression can only match simple one-level patterns.

- All binders receive a unique identifier, there can't be any variable shadowing.

This thesis is not meant to explain the operational semantics of the STG language in its entirety, however, the semantics of let expressions deserves some attention because the evaluation of a let expression is basically equivalent to a heap allocation. The heap is represented as a map storing closures and data constructors under a specific heap address. When the STG-machine encounters a let expression, it allocates the right-hand side of each binding on the heap before continuing with the evaluation of the let body.

Once a right-hand side has been allocated on the heap, it never gets deleted explicitly, so it might be present for the rest of the program's execution (ignoring the garbage collection of an actual implementation). The access via its binder, however, is restricted by its lexical scope: It can only be accessed during the evaluation of the let body or, if the binding is recursive, in the right-hand side itself.

2.5. Escape analysis in the context of Haskell

Since the escape property of a right-hand side depends on the lexical scope of its binding and reassigning bindings is not possible in Haskell, we can consider the binder of an object to be the object's name. It would therefore be possible and very convenient to rather assign the escape property to the binder than to the right-hand side: Considering the binding `let b = Just x`, the two statements "`Just x` *escapes*" and "`b` *escapes*" would be equivalent by this definition, the latter being much simpler. From now on, when stating that a binder b escapes or doesn't escape, we are actually referring to b's right-hand side. The following sections describe significant characteristics and challenges to overcome while developing an escape analysis in Haskell.

2.5.1. The point of using STG for escape analysis

Most analyses that are performed in the GHC, process programs at the level of Core which is a very minimal but still high level representation. However, optimizations that are based on escape analysis usually operate on a relatively low-level: They deal with memory allocation and other inner workings of the programming languages runtime. Therefore, the escape analysis itself should be aware of those low-level features as well. Particularly, if we intend to use the escape analysis to identify objects, that can be allocated on stack, the intermediate program representation should have a concept of memory allocation.

Taking this into account, we would favor the STG language over the Core language for the escape analysis: The STG-machine has the concept of a heap built in and explicitly specifies, when and how to allocate memory while at Core-level, this information is supposed to be transparent. Consider the following code in the STG language:

```
let rec fac n = let n' = n - 1
                    fac' = fac n'
                in case n of    0 -> 1
                                _ -> n * fac'
```

Operationally, the STG-machine would allocate a closure (more specifically, a thunk) for `n'` and `fac'` before evaluating the case expression. This behavior can easily be derived from the STG code, which is beneficial for an escape analysis.

However, the STG code could have been generated from the following Core code:

```
let fac n = case n of   0 -> 1
                        _ -> n * fac (n - 1)
```

Here, the second argument passed to `(*)` is an expression (`fac (n - 1)`) and also the argument passed to `fac` is an expression (`n - 1`). It is not apparent that, in later stages, the compiler would generate a closure allocation for each arguments. This makes the Core language inappropriate for our purposes.

2.5.2. Consideration of the lexical scope

If an object is accessed outside the lexical scope of the variable it is bound to at the object's allocation, it escapes. In the context of **let** $x = e_1$ **in** e_2, the expression e_1 allocates an object which is bound by the variable x. The object escapes if and only if it is referred to outside the lexical scope of x, namely e_1 and e_2.

2.5.3. Challenges caused by closures and thunks

Haskell represents functions as closures at runtime that capture the free variables. This unfortunately makes escape analysis more complicated: If a closure escapes, it could cause their free variables to escape as well. The same problem occurs with thunks which represent values at runtime, that have yet to be evaluated. Thunks

are used in Haskell for the implementation of laziness. In the following STG code, `x` can only be accessed in the outer let body:

```
let x = 42
in  let y = case x of   1 -> True
                        _ -> False
    in  Just y
```

Since `x` is only used in the definition of `y` which is of the type `Bool`, in a strict language `x` wouldn't escape, since booleans can't contain pointers. But since Haskell is lazy, `y` is represented as a thunk at runtime, containing `x` as a free variable. Since `y` escapes without getting evaluated in the lexical scope of `x`, it might get evaluated after the evaluation of `x`'s let body. So we have to classify `x` as escaping as well.

2.5.4. Escape through data structures

Considering a data structure that contains references to further objects, there are additional consequences of the escape of said data structure. For example when analysing the data constructor `Just x`, bear in mind, that if the constructed object is being passed outside `x`'s lexical scopes, it escape as well.

2.5.5. Escape through function calls

Similar to the data constructors, the result of a function application can cause the function's arguments to escape. However, the result doesn't necessarily contain pointers to all the arguments. There are different cases to consider:

saturated call If the function is known at compile time, it is possible to narrow down the arguments that can escape through the call by analysing the function's code first. Keep in mind that this is only valid if the expected amount of arguments is provided, so the function's code can be executed right away.

unsaturated call With currying it is possible to call a function only partially, which means that there are fewer arguments passed to the function than expected. In this case the code of the function doesn't get executed right away, instead a new closure is being constructed that takes the remaining arguments, that are necessary to saturate the call. The arguments passed for an unsaturated call appear as free variables in that closure. The closure resulting from the partial application also contains a reference to the original closure's code and free variables. So if the result of an unsaturated call escapes, both, the arguments passed to the call and the closure itself must be assumed to escape.

unknown call If the concrete function to be called is dynamically bound, it's not generally possible to know if the result has references to the function's arguments. Thus, similarly to data constructors, if the result of an unknown function's application escapes, all arguments have to be assumed to escape as well. Neither

is it possible to know if an unknown call is saturated or not, therefore the function itself has to be assumed to escape as well.

over-saturated call If there are more arguments provided to the function than expected, the expected amount of arguments is being consumed for a saturated call, the excess arguments are applied to the result of the call. While for the saturating arguments the call can be handled identically to a saturated call, the excess arguments are applied to a function, that might not be known at compile time. Those arguments have to be handled as if they were passed for an unknown call.

2.5.6. Challenges caused by join points

Join points are described in [9] to resemble points in the program, where control flow of different branches joins together again. For optimization purposes, join points have been added to the STG definition in the GHC. From a denotational standpoint, join points are equivalent to regular bindings and the syntax of defining join points and calling join points is very similar as well. However, the conditions that must hold for join points allow for more optimized operational semantics: Join points are guaranteed to never be captured in a closure or thunk and always be tail-called with the expected number of arguments. The call to a join point can therefore be compiled to a simple jump, no closure needs to be allocated. Before the jump occurs, the current evaluation context is being discarded, which includes all frames on the stack above the join point. Therefore, special care must be taken, when deciding whether to allocate variables on stack.

Consider the following example:

```
join j = \ y ->  in  case y of  True    -> 1
                                _       -> 2
in   let x = z == 0
     in  j x
```

The join point `j` gets called in the body of the inner let expression. `x` is defined in the caller context and gets passed to `j`. Since the value of `j x` doesn't contain a reference to it, `x` wouldn't escape with the usual let semantics. However, allocating `x` on the stack would lead to problems. Since the frame for `j` would be located lower on the stack than `x`, the jump to `j` would cause `x` to be deleted and could not access it via the argument.

2.5.7. Escape through case expressions

There is generally the possibility for objects to escape through the result of a case expression, namely if the case alternative that is taken at runtime evaluates to an object containing a reference to another object. If in the following example `scrut` evaluates to `True`, the first case will be taken, which causes `x` to escape:

```
case scrut of   True        -> x
                _           -> ...
```

It is also possible that the case scrutinee or at least parts of it escape when the alternative constructor of the case that has been taken binds objects that escape through the right-hand side:

```
case scrut of   Left y   -> y
                z        -> z
```

In this example, there are two possible alternatives. In the first one, `scrut` has a reference to an object which is accessible from the case alternative through the name `y` and, therefore, might escape. In the second case alternative the entire case scrutinee is accessible through the name `z`. Since `z` is being passed outside the lexical scope of `scrut`, it has to be classified as escaping.

In general, it is unknown, which case alternative will be taken at compile time. So every case alternative has to be taken into account for the escape analysis of a case expression.

3. Design and implementation

We are presenting the specification for a flow-sensitive escape analysis algorithm for Haskell. The design decision are elaborated in section 3.2.

3.1. Escape analysis algorithm

Figure 3.1 shows the syntax used to describe the algorithm. We omit parts of the STG language that are either irrelevant in the context of the algorithm or don't contribute to the understanding of the algorithm.

Constructors	$K \in \mathsf{Con}$		
Variables	$a, b, f, x \in \mathsf{Var}$		
Right-hand sides	$r \in \mathsf{Val}$	$::=$	$\lambda\, \overline{a}\, .\, e \mid K\ \overline{a}$
Expressions	$e, s \in \mathsf{Exp}$	$::=$	$\mathbf{let}\ b = r\ \mathbf{in}\ e$
		$\mid$	$\mathbf{let\ rec}\ \overline{b = r}\ \mathbf{in}\ e$
		$\mid$	$\mathbf{join}\ b = r\ \mathbf{in}\ e$
		$\mid$	$\mathbf{join\ rec}\ \overline{b = r}\ \mathbf{in}\ e$
		$\mid$	$f\ \overline{a}$
		$\mid$	$\mathbf{case}\ s\ \mathbf{of}\ \overline{K\ \overline{a} \to e}$
Programs	$P \in \mathsf{Prog}$	$::=$	$\overline{b = r}$
Escaping binders	$B \subseteq \mathsf{Var}$		
Usage classes	$t \in \mathsf{U}$	$::=$	$\mathcal{E} \mid \mathcal{V} \mid \mathcal{R} \mid \mathcal{N}$
Free variable usages	$\phi \in \mathsf{Var} \to \mathsf{U}$		
Usage signatures	$\sigma \in \mathsf{Sig}$	$::=$	$\overline{t}$
Signature environments	$\rho \in \mathsf{Var} \rightharpoonup \mathsf{Sig}$		
Join point environments	$\iota \in \mathsf{Var} \rightharpoonup (\mathsf{Var} \to \mathsf{U})$		
Function arity	$\alpha \in \mathsf{Val} \to \mathbb{N}$		

Figure 3.1.: Syntax

$$\boxed{\mathscr{E}[\![e]\!]^{\iota}_{\rho} = \langle \phi^e, B^e \rangle}$$

$$\mathscr{E}[\![\mathbf{let}\ b = r\ \mathbf{in}\ e]\!]^{\iota}_{\rho} = \begin{array}{ll} \mathit{let} & \langle \phi^r, B^r, \sigma^r \rangle = \mathscr{R}[\![r]\!]^{\iota}_{\rho} \\ & \iota' = \{(x, y[b \mapsto \mathcal{E}]) \mid (x, y) \in \iota\} \\ & \rho' = \rho[b \mapsto \sigma^r] \\ & \langle \phi^e, B^e \rangle = \mathscr{E}[\![e]\!]^{\iota'}_{\rho'} \\ & B^{\mathbf{let}} = \begin{cases} B^r \cup B^e \cup \{b\} & \text{if } \phi^e(b) = \mathcal{E} \\ B^r \cup B^e & \text{else} \end{cases} \\ \mathit{in} & \left\langle \phi^e \lhd_{\phi^e(b)} \phi^r, B^{\mathbf{let}} \right\rangle \end{array} \tag{3.1}$$

$$\mathscr{E}\left[\!\left[\mathbf{let\ rec}\ \overline{b = r}\ \mathbf{in}\ e\right]\!\right]^{\iota}_{\rho} = \begin{array}{ll} \mathit{let} & \rho' = \rho\left[\overline{b \mapsto \mathcal{N}^{\alpha(r)}}\right] \\ & \langle \phi^{\mathbf{let}}, B^{\mathbf{let}}, \rho'' \rangle = \mathscr{B}\left[\!\left[\mathbf{let}\ \overline{b = r}\ \mathbf{in}\ e\right]\!\right]^{\iota}_{\rho', \emptyset} \\ \mathit{in} & \left\langle \phi^{\mathbf{let}}, B^{\mathbf{let}} \cup \left\{ b_k \mid \phi^{\mathbf{let}}(b_k) = \mathcal{E} \right\} \right\rangle \end{array} \tag{3.2}$$

$$\mathscr{E}[\![\mathbf{join}\ b = r\ \mathbf{in}\ e]\!]^{\iota}_{\rho} = \begin{array}{ll} \mathit{let} & \langle \phi^r, B^r, \sigma^r \rangle = \mathscr{R}[\![r]\!]^{\iota}_{\rho} \\ & \iota' = \{(x, y[b \mapsto \mathcal{E}]) \mid (x, y) \in \iota\}\,[b \mapsto \emptyset] \\ & \rho' = \rho[b \mapsto \sigma^r] \\ & \langle \phi^e, B^e \rangle = \mathscr{E}[\![e]\!]^{\iota'}_{\rho'} \\ & B^{\mathbf{let}} = \begin{cases} B^r \cup B^e \cup \{b\} & \text{if } \phi^e(b) = \mathcal{E} \\ B^r \cup B^e & \text{else} \end{cases} \\ \mathit{in} & \left\langle \phi^e \lhd_{\phi^e(b)} \phi^r, B^{\mathbf{let}} \right\rangle \end{array} \tag{3.3}$$

$$\mathscr{E}\left[\!\left[\mathbf{join\ rec}\ \overline{b = r}\ \mathbf{in}\ e\right]\!\right]^{\iota}_{\rho} = \begin{array}{ll} \mathit{let} & \rho' = \rho\left[\overline{b \mapsto \mathcal{N}^{\alpha(r)}}\right] \\ & \langle \phi^{\mathbf{let}}, B^{\mathbf{let}}, \rho'' \rangle = \mathscr{B}\left[\!\left[\mathbf{join}\ \overline{b = r}\ \mathbf{in}\ e\right]\!\right]^{\iota}_{\rho', \emptyset} \\ \mathit{in} & \left\langle \phi^{\mathbf{let}}, B^{\mathbf{let}} \cup \left\{ b_k \mid \phi^{\mathbf{let}}(b_k) = \mathcal{E} \right\} \right\rangle \end{array} \tag{3.4}$$

$$
\begin{array}{lll}
\mathscr{E}\,[\![f\ \overline{a}^n]\!]^\iota_\rho = & \begin{array}{ll} let & \sigma^f = \rho(f) \\ & s = \sigma^f \neq \bot \wedge |\sigma^f| \leq n \\ & \phi^f = \begin{cases} \left\{ \overline{\left(a_j, \sigma^f_j \sqcup \iota(f)(a_j)\right)}^{1 \leq j \leq |\sigma^f|} \right\} \\ \quad , \overline{(a_j, \mathcal{E})}^{|\sigma^f|+1 \leq j \leq n}(f, \mathcal{V}) \end{cases} & \text{if } s \\ & \left\{ \overline{(a, \mathcal{E})}^{n}(f, \mathcal{E}) \right\} & \text{else} \\ in & \left\langle \phi^f, \emptyset \right\rangle \end{array} & (3.5) \\
\\
\mathscr{E}\,[\![\textbf{case } s \textbf{ of } \overline{K\ \overline{a} \to e}]\!]^\iota_\rho = & \begin{array}{ll} let & \langle \phi^s, B^s \rangle = \mathscr{E}[\![s]\!]^\iota_\rho \\ & \phi^{\overline{e}} = \bigsqcup \overline{\begin{array}{ll} let & \langle \phi^e, B^e \rangle = \mathscr{E}[\![e]\!]^\iota_\rho \\ in & \phi^e \end{array}} \\ & t = \bigsqcup \overline{\overline{\phi^{\overline{e}}(a)}} \\ & \phi^{\textbf{case}} = \begin{cases} \phi^{\overline{e}} \lhd_{\mathcal{R}} \phi^s & \text{if } t \sqsubseteq \mathcal{R} \\ \phi^{\overline{e}} \lhd_{\mathcal{V}} \phi^s & \text{else} \end{cases} \\ & B^{\overline{e}} = \bigcup \overline{\begin{array}{ll} let & \langle \phi^e, B^e \rangle = \mathscr{E}[\![e]\!]^\iota_\rho \\ in & B^e \end{array}} \\ in & \langle \phi^{\textbf{case}}, B^s \cup B^{\overline{e}} \rangle \end{array} & (3.6)
\end{array}
$$

$$
\boxed{\mathscr{B}\,[\![\textbf{let } \overline{b = r} \textbf{ in } e]\!]^\iota_{\rho,\phi_0} = \left\langle \phi^{\textbf{let}}, B^{\textbf{let}}, \rho' \right\rangle}
$$

$$
\begin{array}{lll}
\mathscr{B}\,[\![\textbf{let } \overline{b = r} \textbf{ in } e]\!]^\iota_{\rho,\phi_0} = & \begin{array}{ll} let & \overline{\langle \phi^r, B^r, \sigma^r \rangle = \mathscr{R}[\![r]\!]^\iota_\rho} \\ & \iota' = \left\{ \left(x, y\overline{[b \mapsto \mathcal{E}]}\right) \mid (x, y) \in \iota \right\} \\ & \rho' = \rho\overline{[b \mapsto \sigma^r]} \\ & \langle \phi^e, B^e \rangle = \mathscr{E}[\![e]\!]^{\iota'}_{\rho'} \\ & \phi^{\textbf{let}} = (\phi_0 \sqcup \phi^e)\,\overline{\lhd_{\phi^e(b)}\ \phi^r} \\ in & \begin{cases} \left\langle \phi_0, \overline{B^r \cup} B^e, \rho \right\rangle & \text{if } \rho' = \rho \wedge \phi^{\textbf{let}} = \phi_0 \\ \mathscr{B}\,[\![\textbf{let } \overline{b = r} \textbf{ in } e]\!]^\iota_{\rho',\phi^{\textbf{let}}} & \text{else} \end{cases} \end{array} & (3.7) \\
\\
\mathscr{B}\,[\![\textbf{join } \overline{b = r} \textbf{ in } e]\!]^\iota_{\rho,\phi_0} = & \begin{array}{ll} let & \overline{\langle \phi^r, B^r, \sigma^r \rangle = \mathscr{R}[\![r]\!]^\iota_\rho} \\ & \iota' = \left\{ \left(x, y\overline{[b \mapsto \mathcal{E}]}\right) \mid (x, y) \in \iota \right\} \iota'\overline{[b \mapsto \emptyset]} \\ & \rho' = \rho\overline{[b \mapsto \sigma^r]} \\ & \langle \phi^e, B^e \rangle = \mathscr{E}[\![e]\!]^{\iota'}_{\rho'} \\ & \phi^{\textbf{let}} = (\phi_0 \sqcup \phi^e)\,\overline{\lhd_{\phi^e(b)}\ \phi^r} \\ in & \begin{cases} \left\langle \phi_0, \overline{B^r \cup} B^e, \rho \right\rangle & \text{if } \rho' = \rho \wedge \phi^{\textbf{let}} = \phi_0 \\ \mathscr{B}\,[\![\textbf{join } \overline{b = r} \textbf{ in } e]\!]^\iota_{\rho',\phi^{\textbf{let}}} & \text{else} \end{cases} \end{array} & (3.8)
\end{array}
$$

$$\boxed{\mathscr{R}[\![r]\!]^{\iota}_{\rho} = \langle \phi^r, B^r, \sigma^r \rangle}$$

$$\mathscr{R}[\![\lambda\, \overline{a}\, .\, e]\!]^{\iota}_{\rho} = \begin{array}{ll} let & \langle \phi^e, B^e \rangle = \mathscr{E}[\![e]\!]^{\iota}_{\rho} \\ & \phi^{\lambda} = \phi^e \\ in & \left\langle \phi^{\lambda}, B^e, \overline{\phi^e(a)} \right\rangle \end{array} \tag{3.9}$$

$$\mathscr{R}[\![K\ \overline{a}]\!]^{\iota}_{\rho} = \left\langle \left\{ \overline{(a, \mathcal{E})} \right\}, \emptyset, \langle\rangle \right\rangle \tag{3.10}$$

$$\boxed{\mathscr{A}[\![P]\!]^{\iota}_{\rho} = B^P}$$

$$\mathscr{A}[\![b = r]\!]^{\iota}_{\rho} = \begin{array}{ll} let & \langle \phi^r, B^r, \sigma^r \rangle = \mathscr{R}[\![r]\!]^{\iota}_{\rho} \\ in & B^r \end{array} \tag{3.11}$$

$$\mathscr{A}\left[\!\!\left[\overline{b_j = r_j}^{1 \le j \le n} \right]\!\!\right]^{\iota}_{\rho} = \begin{array}{ll} let & \langle \phi^r, B^r, \sigma^r \rangle = \mathscr{R}[\![r_1]\!]^{\iota}_{\rho} \\ & \rho' = \rho[b \mapsto \sigma^r] \\ in & B^r \cup \mathscr{A}\left[\!\!\left[\overline{b_j = r_j}^{2 \le j \le n} \right]\!\!\right]^{\iota}_{\rho'} \end{array} \tag{3.12}$$

$$\mathcal{N} \sqsubseteq \mathcal{R} \sqsubseteq \mathcal{V} \sqsubseteq \mathcal{E} \tag{3.13}$$

$$\sqcup \in (\mathsf{Var} \to \mathsf{U}) \to (\mathsf{Var} \to \mathsf{U}) \to (\mathsf{Var} \to \mathsf{U}) \tag{3.14}$$

$$\left(\phi^e \sqcup \phi^{e'}\right)(b) = \phi^e(b) \sqcup \phi^{e'}(b) \tag{3.15}$$

$$\lhd \in (\mathsf{Var} \to \mathsf{U}) \to (\mathsf{Var} \to \mathsf{U}) \to \mathsf{U} \to (\mathsf{Var} \to \mathsf{U}) \tag{3.16}$$

$$(\phi^e \lhd_t \phi^{e'})(b) = \begin{cases} \mathcal{E} & \text{if } t = \mathcal{E} \wedge \phi^{e'}(b) \neq \mathcal{N} \\ (\phi^e \sqcup \phi^{e'})(b) & \text{if } t = \mathcal{V} \\ \phi^e(b) \sqcup \mathcal{R} & \text{if } t = \mathcal{R} \wedge \phi^{e'}(b) \neq \mathcal{N} \\ \phi^e(b) & \text{else} \end{cases} \tag{3.17}$$

3.2. Exploring design decisions

The following sections elaborate on the formal definition of the algorithm and how to overcome the challenges identified in section 2.5. First, some general design decisions have to be made:

- For the reasons mentioned in section 2.5.1, the algorithm implemented as part of this thesis operates on the level of the STG representation.
- Most memory allocations in STG are caused by let expressions. There are several other ways to allocate memory (e.g. primitive operations), but to simplify the analysis it is reasonable to only look at let bound objects.

- When analysing a function, the algorithm doesn't take into account the call site, it is therefore *context insensitive*. While a context-sensitive algorithm is expected to yield better results it would increase the complexity considerably and exceed the time limitations for this thesis.

3.2.1. Escape of bound variables

Since the STG language is based on composing expressions, we design a function $\mathscr{E}[\![e]\!] = B^e$ that analyses the STG code on the expression level and returns a set containing all binders B^e that escape through e. Note, that for variables that are free in e this is not generally possible, because the lexical scope of a free variable is not entirely visible on the expression level. When, for example, analysing the expression `Just x` where `x` is a free variable, it is unclear if $B^{\texttt{Just x}}$ should contain `x` or not. Thus, to define B^e as the set that contains the variables that escape through e, B^e can only contain the bound variables of e because for bound variables the entire lexical scope is visible.

3.2.2. Usage of free variables

On the other hand when, analysing a let expression $\mathbf{let}\ b = r\ \mathbf{in}\ e$ where b is a bound variable and the lexical scope is therefore visible, determining whether b should be in $B^{\mathbf{let}}$ requires more detailed knowledge about e. Since b can be a free variable in e, $\mathscr{E}$ should also provide information about the usage of free variables in the expression to analyse.

This is why $\mathscr{E}[\![e]\!]$ has to return a tuple $\langle \phi^e, B^e \rangle$, where ϕ^e is defined as the set of free variables, that escape from the expression e. To demonstrate how this can be useful, attempt to analyse the previous let expression, in the following referred to as **let**. The escaping bindings of the entire let expression $B^{\mathbf{let}}$ can be defined as either $B^r \cup B^e$ or $B^r \cup B^e \cup \{b\}$, depending on whether b escapes or not. b can be a free variable in r and e. To determine if b escapes (if $b \in B^{\mathbf{let}}$ holds), e has to be analysed and only if b is in ϕ^e, we add b to $B^{\mathbf{let}}$.

But now the problem arises, how to construct the free variable usage for the entire let expression $\phi^{\mathbf{let}}$. The intuitive idea would be to define it as $(\phi^r \cup \phi^e) \setminus \{b\}$ (note that b is not free in **let**). So the analysis of let expressions would be defined as

$$\mathscr{E}[\![\mathbf{let}\ b = r\ \mathbf{in}\ e]\!] = \begin{array}{ll} let & \langle \phi^r, B^r \rangle = \mathscr{E}[\![r]\!] \\ & \langle \phi^e, B^e \rangle = \mathscr{E}[\![e]\!] \\ in & \begin{cases} \langle (\phi^r \cup \phi^e) \setminus \{b\}, B^r \cup B^e \cup \{b\} \rangle & \text{if } b \in \phi^e \\ \langle (\phi^r \cup \phi^e) \setminus \{b\}, B^r \cup B^e \rangle & \text{else} \end{cases} \end{array}.$$

Unfortunately this would not detect escapes caused by closures, which are described in section 2.5.3. In the following examples the inner let expression will be referred to as **let**, the right-hand side as r (`\ x -> case`...) and the let body as e (`f`).

```
let a = z == 0
in  let f = \ x -> case a of    True    -> 1
                               _       -> 2
    in  f
```

Here `f` escapes and since it is represented as a closure which contains `a` as a free variable, `a` escapes as well. The approach introduced in the last section would be able to detect the escape of `y` but not the escape of `a`: ϕ^r doesn't contain `a` and the information about `a` being referred to by the closure of `y` gets lost, To prevent that, $\mathscr{E}$ must provide that information as well.

One way to accomplish that, is to represent ϕ^e as a function that specifies different *usage classes* for every binding b that is used in an expression e. These usage classes are:

$\mathcal{E}$: The value of e may contain b, which enables b's right-hand side to be evaluated after the evaluation of e. Thus, the value of e indirectly contains pointers to each free variable of b's right-hand side. For example, in the context of `Just b`, `b` is classified as $\mathcal{E}$.

$\mathcal{R}$: The value of e neither contains b nor any pointers coming from $b's$ return value. The usage of b doesn't release any pointers to its free variables in e. This is the case when b is just being scrutinized in a case expression. For example, in the context of `case b of _ -> Nothing`, `b` is classified as $\mathcal{R}$.

$\mathcal{N}$: b is not being used in e at all ($\mathcal{N}$ is the $\bot$ element: If ϕ is not explicitly defined for b, we implicitly define $\phi^e(b) = \mathcal{N}$).

Now analysing the previous example shall lead to the following results:

$$\phi^r = \{(\texttt{x}, \mathcal{R})\} \qquad \phi^e = \{(\texttt{f}, \mathcal{E})\} \qquad \phi^{\mathbf{let}} = \{(\texttt{x}, \mathcal{E})\}$$
$$B^r = \emptyset \qquad B^e = \emptyset \qquad B^{\mathbf{let}} = \{\texttt{f}\}$$

The information provided by ϕ^r that `x` is being used as a free variable in the right-hand side of the let expression makes it possible to detect, that returning `y` in the let body leads to the escape of `x`.

To construct $\phi^{\mathbf{let}}$, a new operator $\lhd_t$ is necessary that combines ϕ^r and ϕ^e, where ϕ^r is the usage of the free variables in the right-hand side of a binder b', ϕ^e is the usage of free variables in the lexical scope of b' and t is the usage of b' in e ($t = \phi^e(b')$). It is defined as:

$$\mathcal{N} \sqsubseteq \mathcal{R} \sqsubseteq \mathcal{E}$$

$$(\phi^e \sqcup \phi^r)(b) = \phi^e(b) \sqcup \phi^r(b)$$

$$(\phi^e \lhd_t \phi^r)(b) = \begin{cases} \mathcal{E} & \text{if } t = \mathcal{E} \wedge \phi^r(b) \neq \mathcal{N} \\ \phi^e(b) \sqcup \mathcal{R} & \text{if } t = \mathcal{R} \wedge \phi^r(b) \neq \mathcal{N} \\ \phi^e(b) & \text{else} \end{cases}$$

The different cases of $\lhd_t$ are explained below:

$t = \mathcal{E} \wedge \phi^r(b) \neq \mathcal{N}$**:** The value of e may contain a pointer to a binder b'. Thus, if b is used as a free variable in r, the value of the expression **let** $b' = r$ in e may contain a pointer to b.

$t = \mathcal{R} \wedge \phi^r(b) \neq \mathcal{N}$**:** b' may be used as a free variable in e but the value of e cannot contain a pointer to b'. Thus, if another binder b is used as a free variable in r, it is consequently used as a free variable in the expression **let** $b' = r$ in e (at least $\mathcal{R}$).

else: b' is not a free variable in e. Thus, whether another binder b is used in the expression **let** $b' = r$ in e, solely depends on the usage of free variables in e.

Based on this operator, $\mathscr{E}$ cold be defined for non-recursive let expressions as follows:

$$\mathscr{E}[\![\textbf{let } b = r \textbf{ in } e]\!] = \begin{array}{ll} let & \langle \phi^r, B^r \rangle = \mathscr{E}[\![r]\!] \\ & \langle \phi^e, B^e \rangle = \mathscr{E}[\![e]\!] \\ & B^{\textbf{let}} = \begin{cases} B^r \cup B^e \cup \{b\} & \text{if } \phi^e(b) = \mathcal{E} \\ B^r \cup B^e & \text{else} \end{cases} \\ in & \left\langle \phi^e \lhd_{\phi^e(b)} \phi^r, B^{\textbf{let}} \right\rangle \end{array}$$

Keeping track of the usage of all free variables of an expression makes it possible to detect all escaping objects. However, it is still not clear how to treat the following case. This example shows the variable in question `f` being a function that is called:

```
let f = \ x -> case a of     True     -> Just b
                             _        -> Just x
in  f y
```

The value of the let body doesn't contain a pointer to `f` because it is being entered immediately, so it can't be evaluated outside the let expression. Hence, classify `f` as $\mathcal{E}$ might be too conservative. Analysing the let expression would lead to the following result:

$$\begin{array}{lll} \phi^r = \{(\texttt{a}, \mathcal{R}), (\texttt{b}, \mathcal{E})\} & \phi^e = \{(\texttt{f}, \mathcal{E}), (\texttt{y}, \mathcal{E})\} & \phi^{\textbf{let}} = \{(\texttt{a}, \mathcal{E}), (\texttt{b}, \mathcal{E}), (\texttt{y}, \mathcal{E})\} \\ B^r = \emptyset & B^e = \emptyset & B^{\textbf{let}} = \{\texttt{f}\} \end{array}$$

This is not really optimal since it is apparent that the value of `f y` may contain a pointer to the free variable `b` but not to `a`. $\phi^{\textbf{let}}(\texttt{a})$ should therefore be classified as $\mathcal{R}$. The call to `f` is saturated and its closure gets entered right away, which leads to the evaluation of the case expression. Also `f` doesn't escape because can only be used in the let body.

But classifying `f` as $\mathcal{R}$ actually not be sound because we would ignore the fact, that the value of `f y` contains a pointer to either `b` or `y`.

To distinguish the usage of `a` and `f`, another usage class can be introduced:

$\mathcal{V}$: The usage of a binder b in the context of an expression e is $\mathcal{V}$ if the value of e doesn't contain b. So b's right-hand side can only be accessed during the evaluation of e. However, the value of e may have a pointer to the return value of b. This is the case, if e represents a saturated call to b. For example, in the context of `b x` where `b` expects exactly one argument, `b` is classified as as $\mathcal{V}$.

$$\mathcal{N} \sqsubseteq \mathcal{R} \sqsubseteq \mathcal{V} \sqsubseteq \mathcal{E}$$

$$(\phi^e \lhd_t \phi^{e'})(b) = (\phi^e \sqcup \phi^{e'})(b) \quad \text{if } t = \mathcal{V}$$

Now when analysing a saturated call to a free variable `f`, we classify `f` as $\mathcal{V}$

$$\phi^r = \{(\mathtt{a}, \mathcal{R}), (\mathtt{b}, \mathcal{E})\} \qquad \phi^e = \{(\mathtt{f}, \mathcal{V}), (\mathtt{y}, \mathcal{E})\} \qquad \phi^{\mathbf{let}} = \{(\mathtt{a}, \mathcal{R}), (\mathtt{b}, \mathcal{E}), (\mathtt{y}, \mathcal{E})\}$$
$$B^r = \emptyset \qquad B^e = \emptyset \qquad B^{\mathbf{let}} = \emptyset$$

This improves the accuracy of the result in that we correctly detect, that the value of the application of `f` doesn't contain a pointer to `a` and also that `f` doesn't escape.

3.2.3. Analysing function definitions and applications

For saturated calls, we distinguish the usage of the different arguments passed to the function. To do so, we need to analyse the entry code of the function first and examine the usage of the arguments resulting in a *usage signature*. The analysis of a function definition r should result in something like $\langle \phi^r, B^r, \sigma^r \rangle$ where σ^r is the usage signature of the function r. The usage signature is defined as a sequence of usages, where σ^r_n is the usage of the nth argument passed to r. For example, in the context of `f x y z = case y of _ -> Just z`, $\rho(\mathtt{f})$ is $\mathcal{NRE}$.

A lambda definition $\lambda\,\overline{a}\,.\,e$ is analysed by a function $\mathscr{R}[\![\lambda \overline{a} \mapsto e]\!]$, which evaluates the functions body first and look at the usage of each function argument $\overline{a}$ in the body to construct the usage signature:

$$\mathscr{R}[\![\lambda\,\overline{a}\,.\,e]\!] = \begin{array}{ll} let & \langle \phi^e, B^e \rangle = \mathscr{E}[\![e]\!] \\ in & \left\langle \phi^e, B^e, \overline{\phi^e(a)} \right\rangle \end{array}$$

For the call site of that function we want to use that usage signature to examine, how the arguments passed to the function are being used. To have access to the usage signature at the call site, a signature environment $\rho \in \mathsf{Var} \rightharpoonup \mathsf{Sig}$ is necessary, that is being passed to $\mathscr{E}$ and $\mathscr{R}$ and keeps track of the usage signatures of all binders that are in-scope.

When analysing a let expression, we analyse the right-hand side r of a let binding first. If r is a function, σ^r is the usage signature, which gets added to the signature

environment. For analysing the let body, we pass the modified signature environment ρ' in order have access to the usage signature when analysing the call site:

$$\mathscr{E}[\![\textbf{let } b = r \textbf{ in } e]\!]_\rho = \begin{array}{ll} let & \langle \phi^r, B^r, \sigma^r \rangle = \mathscr{R}[\![r]\!]_\rho \\ & \rho' = \rho[b \mapsto \sigma^r] \\ & \langle \phi^e, B^e \rangle = \mathscr{E}[\![e]\!]_{\rho'} \\ & B^{\textbf{let}} = \begin{cases} B^r \cup B^e \cup \{b\} & \text{if } \phi^e(b) = \mathcal{E} \\ B^r \cup B^e & \text{else} \end{cases} \\ in & \left\langle \phi^e \lhd_{\phi^e(b)} \phi^r, B^{\textbf{let}} \right\rangle \end{array}$$

When analysing a function call, we first look up the function in the signature environment. If the function is not in the environment (unknown call), we need to assume every argument passed to the call to escape. Otherwise, by looking at the number of elements in the signature, it is possible to identify whether the call is saturated, unsaturated or over-saturated and treat the arguments accordingly as described in section 2.5.5:

$$\mathscr{E}\,[\![f\ \overline{a}^n]\!]_\rho = \begin{array}{ll} let & \sigma^f = \rho(f) \\ & s = \sigma^f \neq \bot \wedge |\sigma^f| \leq n \\ & \phi^f = \begin{cases} \left\{ \overline{\left(a_j, \sigma^f_j\right)}^{1 \leq j \leq |\sigma^f|}, \overline{(a_j, \mathcal{E})}^{|\sigma^f|+1 \leq j \leq n}(f, \mathcal{V}) \right\} & \text{if } s \\ \left\{ \overline{(a, \mathcal{E})}^n (f, \mathcal{E}) \right\} & \text{else} \end{cases} \\ in & \left\langle \phi^f, \emptyset \right\rangle \end{array}$$

3.2.4. Analysing case expressions

First, we analyse the case scrutinee: $\langle \phi^s, B^s \rangle = \mathscr{E}[\![s]\!]_\rho$. While any of the branches of a case expression can be chosen at runtime we have to merge the free variable usages of all branches:

$$\phi^{\overline{e}} = \bigsqcup \overline{\begin{array}{ll} let & \langle \phi^e, B^e \rangle = \mathscr{E}[\![e]\!]_\rho \\ in & \phi^e \end{array}}$$

Since the scrutinee gets matched against the alt constructors $\overline{K\ \overline{a}}$ and the binders $\overline{\overline{a}}$ of those constructors can be used in the branches $\overline{e}$ of the case expression, we define the usage t of the case scrutinee as the supremum of the usages of the alt constructor binders:

$$t = \bigsqcup \overline{\overline{\phi^{\overline{e}}(a)}}$$

If none of the branches' values contain pointers to the scrutinees or its attributes ($t = \mathcal{R}$), the value of the entire case expression can't contain a pointer to the scrutinee

or its attributes. Hence we merge $\phi^{\overline{e}}$ and ϕ^s with $\lhd_{\mathcal{R}}$ in this case. Otherwise, we use $\lhd_{\mathcal{V}}$:

$$\phi^{\mathbf{case}} = \begin{cases} \phi^{\overline{e}} \lhd_{\mathcal{R}} \phi^s & \text{if } t \sqsubseteq \mathcal{R} \\ \phi^{\overline{e}} \lhd_{\mathcal{V}} \phi^s & \text{else} \end{cases}$$

Note, that s can't be a thunk or a closure, which is why the result of the case expression can't contain a pointer to the scrutinees free variables unless their usage is $\mathcal{E}$ That's why it is possible to use $\lhd_{\mathcal{V}}$ instead of $\lhd_t$.

3.2.5. Analysing recursive bindings via fixed-point iteration

Since bindings can be recursive, it is not generally possible to always analyse the definition of a variable before its usage. The definition of a recursive binding is part of its own lexical scope. When analysing recursive let bindings, we don't only have to examine how the binding is used in the let body, but also how it is used in all of the right-hand sides. In this example three bindings refer to each other recursively:

```
let a = 2:b
    b = 0:c
    c = 4:a
in  c
```

$$\phi^{\texttt{2:b}} = \{(\texttt{b}, \mathcal{E})\} \qquad \phi^{\texttt{0:c}} = \{(\texttt{c}, \mathcal{E})\} \qquad \phi^{\texttt{4:a}} = \{(\texttt{a}, \mathcal{E})\} \qquad \phi^{\texttt{c}} = \{(\texttt{c}, \mathcal{E})\}$$

If we merge the free variable usage of every right-hand side sequentially in the order `2:b`, `0:c`, `4:a` with the let body, we would get the following result:

$$\begin{aligned} \phi' &:= \phi^{\texttt{c}} \lhd_{\phi \texttt{c}(\texttt{a})} \phi^{\texttt{2:b}} = \{(\texttt{c}, \mathcal{E})\} \lhd_{\mathcal{N}} \{(\texttt{b}, \mathcal{E})\} = \{(\texttt{c}, \mathcal{E})\} \\ \phi'' &:= \phi' \lhd_{\phi'(\texttt{b})} \phi^{\texttt{0:c}} = \{(\texttt{c}, \mathcal{E})\} \lhd_{\mathcal{N}} \{(\texttt{c}, \mathcal{E})\} = \{(\texttt{c}, \mathcal{E})\} \\ \phi''' &:= \phi'' \lhd_{\phi''(\texttt{c})} \phi^{\texttt{4:a}} = \{(\texttt{c}, \mathcal{E})\} \lhd_{\mathcal{E}} \{(\texttt{a}, \mathcal{E})\} = \{(\texttt{a}, \mathcal{E}), (\texttt{c}, \mathcal{E})\} \\ B^{\mathbf{let}} &= \{\texttt{a}, \texttt{c}\} \end{aligned}$$

Processing the bindings in this order, we fail to detect the escape of `b`. This is why a fixed-point iteration is necessary to analyse recursive bindings.

It also might happen that, when analysing a recursive function call, the function's usage signature might not be known yet. In the following example, the first call to `f` that must be analysed is the recursive call. But the usage signature for `f` can't be known at that point:

```
let f = \ x y -> case x > y of  True    -> x : f (x + 1) y
                               False   -> []
in  f a b
```

There must be a convention on how to treat known functions, of which the usage signature is not known for the analysis of the call site. Two possible methods would be:

All $\mathcal{E}$

We initially assume the usage signature of a known function f with the arity N to be $\sigma^f = \mathcal{E}^N$ (worst case). Analysing the following example shows that this method would actually yield unsatisfactory results:

```
let f = \ x y -> case x > y of  False   -> Just x
                                True    -> f (x + 1) y
in  f a b
```

Note, that the return value of `f` has a pointer to the first argument at most. The second argument is solely used in the case scrutinee and is always passed to the recursive call as second argument again. So ideally the usage signature should be identified as $\sigma^{\mathtt{f}} = \mathcal{ER}$.

Before we analyse the recursive call of `f` we modify the signature environment to be $\rho' = \rho[\mathtt{f} \mapsto \mathcal{EE}]$. It is apparent that passing `y` again as the second argument to the recursive call would cause the final usage signature to still be $\sigma^{\mathtt{f}} = \mathcal{EE}$. Even iterating the analysis of `f`'s definition wouldn't lead to any improvement since the usage signature didn't change, and thus we already reached a fixed point.

All $\mathcal{N}$

We initially assume the usage signature of a known function f with the arity N to be $\sigma^f = \overline{\mathcal{N}}^N$ (best case). Using this method to analyse the previous example would lead to the desired result, but the next one shows an instance, where multiple iterations are necessary:

```
let f = \ x y -> case x > y of  False   -> Just x
                                True    -> f y (x + 1)
in  f a b
```

Here the arguments of the recursive call are swapped, so when the **`False`** branch is taken, the value of `f` contains the first argument, if the **`True`** branch is taken, the value contains a pointer to the second one. So the usage signature should be $\sigma^{\mathtt{f}} = \mathcal{EE}$. We get that result after two iterations.

The following example proves that for arbitrary recursive definitions, the number of necessary iterations depends on the maximum number of arguments:

```
let f = \ x1 ... xN -> case ... of
                         ... -> Just x1
                         ... -> f x2 ... xN x1
in  f z1 ... zN
```

After the first iteration, the usage signature would be $\sigma^{\mathtt{f}} = \mathcal{E}\overline{\mathcal{N}}^{N-1}$. But since in the recursive call the arguments are rotated by one, in the second iteration we would get $\sigma^{\mathtt{f}} = \mathcal{EE}\overline{\mathcal{N}}^{N-2}$. This pattern continues with the nth iteration resulting in $\sigma^{\mathtt{f}} = \overline{\mathcal{E}}^n\overline{\mathcal{N}}^{N-n}$.

The fixed-point algorithm used for the analysis of recursive bindings is explained in the following. Before analysing recursive bindings, the signature environment has

to be extended by an initial usage signatures (all $\mathcal{N}$) for every binding. We delegate the analysis of the binders and the let body to a separate function $\mathscr{B}$ which performs the fixed-point iteration. Besides the signature environment, $\mathscr{B}$ expects the free variable usage from the previous iteration ϕ_0, which is initialized with $\emptyset$:

$$\begin{aligned}\mathscr{E}\left[\!\left[\textbf{let } \overline{b = r} \textbf{ in } e\right]\!\right]_\rho = \quad & let \quad \rho' = \rho\left[\overline{b \mapsto \overline{\mathcal{N}}^{\alpha(r)}}\right] \\ & \quad\;\; \langle \phi^{\textbf{let}}, B^{\textbf{let}}, \rho' \rangle = \mathscr{B}\left[\!\left[\textbf{let } \overline{b = r} \textbf{ in } e\right]\!\right]_{\rho',\emptyset} \\ & in \quad \left\langle \phi^{\textbf{let}}, B^{\textbf{let}} \cup \left\{ b_k \mid \phi^{\textbf{let}}(b_k) = \mathcal{E} \right\} \right\rangle\end{aligned}$$

In contrast to the analysis of non-recursive binding, $\mathscr{B}$ additionally merges ϕ^e with the result of the previous iteration ϕ_0. At the end of every iteration, $\mathscr{B}$ checks if there are changes in the signature environment or in the free variable usage, in which case $\mathscr{B}$ proceeds with another iteration.

$$\begin{aligned}\mathscr{B}\left[\!\left[\textbf{let } \overline{b = r} \textbf{ in } e\right]\!\right]_{\rho,\phi_0} = \quad & let \quad \overline{\langle \phi^r, B^r, \sigma^r \rangle = \mathscr{R}[\![r]\!]_\rho} \\ & \quad\;\; \rho' = \rho[\overline{b \mapsto \sigma^r}] \\ & \quad\;\; \langle \phi^e, B^e \rangle = \mathscr{E}[\![e]\!]_{\rho'} \\ & \quad\;\; \phi^{\textbf{let}} = (\phi_0 \sqcup \phi^e)\, \overline{\lhd_{\phi^e(b)}\, \phi^r} \\ & in \quad \begin{cases} \left\langle \phi_0, \overline{B^r \cup} B^e, \rho \right\rangle & \text{if } \rho' = \rho \wedge \phi^{\textbf{let}} = \phi_0 \\ \mathscr{B}\left[\!\left[\textbf{let } \overline{b = r} \textbf{ in } e\right]\!\right]_{\rho',\phi^{\textbf{let}}} & \text{else} \end{cases}\end{aligned}$$

3.2.6. Considering join points

As described in section 2.5.6, a jump to a join point would cause all objects that have been stack-allocated above the join point's stack frame to be deleted. This leads to a problem if one of these objects is being passed to the join point as an argument. To allow non-escaping binders to allocate their objects on the stack, we assume an object to escape if it is being passed to a join point which is defined outside its binder's lexical scope.

To accomplish that, another environment $\iota \in \mathsf{Var} \rightharpoonup (\mathsf{Var} \to \mathsf{U})$ is necessary, which keeps track of all bindings, that are defined inside of the lexical scope of each join point. $\iota(f)(b) = \mathcal{E}$ means, that a binder b is defined in the lexical scope of a join point f. Otherwise, $\iota(f)(b)$ returns $\mathcal{N}$ which is the bottom element. When analysing a let binding **let** $b = r$ **in** e, we have to add b to each join point currently in the environment:

$$\iota' = \{(x, y[b \mapsto \mathcal{E}]) \mid (x, y) \in \iota\}$$

When analysing a join point definition **join** $f = r$ **in** e, we also add the join point to the environment.

$$\iota' = \{(x, y[b \mapsto \mathcal{E}]) \mid (x, y) \in \iota\}\,[f \mapsto \emptyset]$$

We use this information during the analysis of a jump to a join point. When a join point f is being jumped to, every binder that is defined inside the lexical scope of f

goes out of scope. If any of those binders are being passed as arguments to f, they escape. To account for that, it is sufficient to define the usage class of those binders to be $\mathcal{E}$ since jumps to join points only occur in tail contexts by definition.

When analysing a jump to a join point f with the arguments $\overline{a}$ and the usage signature σ^f, we construct the free variable usage as follows:

$$\phi^f = \left\{ \overline{\left(a_j, \sigma^f_j \sqcup \iota(f)(a_j)\right)}^{1 \leq j \leq |\sigma^f|} \right\}$$

3.2.7. Challenges caused by side effects

In STG primitive operations are allowed to cause side effects. There is for example an operation `putMVar# :: MVar# s a -> a -> State# s -> State# s`, which writes a value of type `a` into a mutable variable of type `MVar# s a` that is accessible from the current thread's environment.

Note, that a constructor call like `Just x` makes its argument `x` only escape, if its value is demanded outside its lexical scope. But even classifying `x` with $\mathcal{E}$ wouldn't be sufficient for the prim-op call `putMVar# var x`, which would make `x` accessible from potentially anywhere in the thread. This can be shown with the following example:

```
makeXEscape = \ ... ->  let x = createObj ...
                        in  case putMVar# var x of
                                _ -> True

accessXAfterEscape = \ ... -> case readMVar# var of ...
```

If `putMVar#` had no side effects, `x` could not escape even if its usage signature was $\mathcal{E}$, since it is only used in the case scrutinee. But actually, `x` gets put in the mutable variable `var` and can therefore be accessed by the second function.

To solve this problem another usage class $\mathcal{S}$ is introduced:

$$\mathcal{N} \sqsubseteq \mathcal{R} \sqsubseteq \mathcal{V} \sqsubseteq \mathcal{E} \sqsubseteq \mathcal{S}$$

$$(\phi^e \lhd_t \phi^{e'})(b) = \begin{cases} \mathcal{S} & \text{if } \phi^e(b) \sqcup \phi^{e'}(b) = \mathcal{S} \\ \mathcal{E} & \text{if } t = \mathcal{E} \wedge \phi^{e'}(b) \neq \mathcal{N} \\ (\phi^e \sqcup \phi^{e'})(b) & \text{if } t = \mathcal{V} \\ \phi^e(b) \sqcup \mathcal{R} & \text{if } t = \mathcal{R} \wedge \phi^{e'}(b) \neq \mathcal{N} \\ \phi^e(b) & \text{else} \end{cases}$$

Arguments that are being passed to a prim-op, which potentially writes them into mutable memory areas, have to be classified with $\mathcal{S}$. The escape signature of those prim-ops has to be $\mathcal{S}^N$ where N is the arity of the prim-op.

4. Evaluation

The following sections explain how to instrument a Haskell runtime environment for evaluation purposes. The most important findings that can be obtained from that are concerning the effectiveness and the soundness of the escape analysis. Replacing heap allocations with stack allocation is the most common optimization that can be based on escape analysis. Therefore, the effectiveness of an escape analysis can be assessed by determining the amount of stack allocation, that is due to the escape analysis.

Note, that just statically counting the number of non-escaping bindings in the analysed program code isn't representative: In an imperative program, the number of executions per statement are not distributed equally, because the control flow of most programs contains branches and loops. The same goes for the evaluation of expressions in a functional program, which is why some bindings tend to be allocated more often than others. Another problem is that the number of bytes of an allocation may depend on the program's input. It is therefore necessary to measure the amount of allocated memory for each binding at runtime.

4.1. Metrics

The question arises, how to define metrics that can assess the soundness and the effectiveness of an escape analysis for a specific program execution. These metrics have to be conclusive, but we also should be able to calculated then in a feasible manner. Since no optimization has been implemented as part of this thesis, it wouldn't be possible to measure the speedup or reduction of total memory allocation.

On the other hand, we have to recognize that even the effectiveness can't be assessed statically, we can't determine generally, how many times a binding will be allocated at runtime. This problem can be illustrated by the following example:

```
f1 x1 n1 y1 =   let y1' = 2 * y1
                    n1' = n1 + 1
                in case y < x1 of
                    True -> n1
                    _    -> f1 x1 n1' y1'

f2 x2 y2 =      let y2' = 2 * y2
                    f2' = f2 x2 y2'
                in case y2 < x2 of
                    True -> [y2']
```

```
    _     -> y2' : f2'
```

Analysing `f1` shows that `y1'` can be allocated on the stack. In `f2`, nothing can be allocated on the stack. Therefore, `f1` would benefit more from an escape analysis than `f2`. But in the context of a whole program, the benefit depends heavily on the usage profile of `f1` and `f2`. More specifically, in the context of the `main`-function

```
main = do   [a, b] <- getArgs
            print $ f1 (read a) 1 1
            print $ f2 (read b) 1
```

, it depends on the command line arguments: The benefit would be higher when the program is called with `1000000 10` compared to `10 1000000`. With this in mind, it seems unavoidable to make use of profiling functionality for the evaluation.

A conclusive way to evaluate the effectiveness would be to measure the number of allocations while running the program. Even more interesting to measure would be the amount of memory, that could have been allocated on the stack during the runtime.

The main metric we are interested in, is the portion of heap-allocated memory, that could have been allocated on the stack instead. We are able to measure this value under two different assumptions:

S: Given a binding $b = r$ in a program P, we assume that an optimizer replaces the heap allocations of r with a stack allocation, if the escape analysis decides, that the binding doesn't escape $\left(b \notin \mathscr{A}[\![P]\!]^{\emptyset}_{\emptyset}\right)$. Making this assumption allows measuring the effectiveness of the escape analysis for the use of stack allocation.

S^*: Instead of relying on information provided by a static escape analysis, we imagine an optimizer, which has information about the future: It knows about the execution path that will be taken during the execution of the program. Given a binding $b = r$, we assume that an optimizer replaces the heap allocations of r with a stack allocation if none of r's runtime instances will be accessed outside b's lexical scope on the specific execution path taken during the measurement. S^* represents an upper bound for S: Assuming the algorithm to be sound, we expect $S \leq S^*$. Note, that depending on the path coverage, the value for S^* can potentially be much higher than S could ever be for a sound algorithm.

To assess the soundness of the escape analysis, we are estimating the set of false negatives. These come in the form of escaping binders, which aren't detected by the escape analysis:

F_s is a set that contains a binding $b = r$ if and only if any of r's runtime instances is accessed outside the lexical scope of b on the specific execution path taken during the measurement despite being classified as non-escaping. This set is expected to be empty for a sound escape analysis algorithm.

We also might want to be able to identify potential false positives or missed opportunities. These would provide us direction for further improvement of the escape analysis algorithm:

F_h is a set that contains a binding $b = r$ if and only if any of r's runtime instances is not accessed outside the lexical scope of b on the specific execution path taken during the measurement despite being classified as escaping. Every binding in this set potentially represents a missed opportunity of the escape analysis.

How these metrics are calculated is explained in appendix A.1.

4.2. Tracking escaping objects at runtime by instrumenting a Haskell interpreter

To gather the metrics introduced in the last section, it would be desirable to base the measurements on machine code produced by the GHC. The *ticky-ticky profiling* functionality of the GHC provides features that might be helpful for these purposes. It enables Haskell developers to count several kinds of events that happen during a program's execution by injecting the code with instructions that increment certain counters [10]. Defining new counters for further kinds of events is relatively simple by customizing the GHC's runtime environment. This would make it possible to count the number of allocations for non-escaping bindings that are being performed and with some more effort, the amount of memory for these allocations.

A much bigger challenge is to catch the event of a binding going out of scope. The reason for this is that the GHC produces machine code that is in *continuation passing style*. An expression doesn't necessarily return after its evaluation but rather jumps to a continuation. So locating the exact location in the machine code, where a binding runs out of scope is quite challenging.

We also want to keep track of bindings that are currently in scope which requires the use of more complex data structures besides simple counters. Ticky-ticky isn't designed for this purpose. Also, keep in mind that relying on data types defined in Haskell is not an option because those data structures would have to be allocated on the managed heap and would ultimately interfere with our measurements.

The *GHC whole program compiler* is a customized version of the GHC that is able to serialize different intermediate representations and linker metadata and export it to a file for further analysis [11]. The STG code from such a file can also be executed by the *external STG interpreter*. These tools allow for more sophisticated program analyses by simply extending the interpreter state by new data structures and modifying them during runtime. The GHC whole program compiler and the external STG interpreter seem to be adequate tools for the evaluation of the escape analysis.

The external STG interpreter is being enhanced by profiling functionality to validate the results of the escape analysis evaluate its effectiveness and potential. The measurements focus on the use case of allocating non-escaping binders on stack. Several benchmarks programs from the *nofib* benchmark suite [12] are being executed in order to perform these measurements.

4.3. Measurement

Based on the set of defined binders D in the code of a benchmark P with $D \subseteq \text{Var}$ and the set of escaping binders B resulting from the escape analysis $B = \mathscr{A}[\![P]\!]^{\emptyset}_{\emptyset}$ with $B \subseteq D$, following data is being collected as part of the profiler state Γ during the execution of said benchmark:

A^*_h: Binders whose objects actually have been accessed outside their lexical scope during the execution of P

$$A^*_h \subseteq D$$

n: Estimated amount of allocated memory in bytes for a specific binder b during the execution of P

$$n \in D \to \mathbb{N}$$

The profiler state Γ is defined as

$$\begin{aligned} \Gamma &= (A^*_h, n) \\ \Gamma_{\text{init}} &= (A^*_h = \emptyset, n = n_{\text{init}}) \text{ with } n_{\text{init}}(b) = 0 \end{aligned}$$

Not part of the profiler state but passed as separate parameters are:

l: Set of all heap addresses, that didn't exceed the lexical scope of their binder at the current interpreter state ($l \subseteq \mathsf{Addr}$) If an address is accessed at runtime while it is not in l, the object escaped.

j: Mapping from an address of a join point that is currently in-scope to the l-state at the join point's allocation. $\left(j \in \mathsf{Addr} \rightharpoonup 2^{\mathsf{Addr}}\right)$

Pseudo code is used to show the modifications made to the interpreter to perform the measurements. For this illustration we reduce the complexity of the interpreter state to a minimum, leaving out every detail that isn't directly affected by said modifications. The relevant parts of the interpreter state Σ are

$$\Sigma = (\texttt{heap}, \texttt{env})$$

`heap` The heap is represented as a mapping from an address to an object that is stored at this address ($\texttt{heap} \in \mathsf{Addr} \to \mathsf{Obj}$).

`env` The environment keeps track of all binders that are in scope at the current state and what heap object they point to ($\texttt{env} \in D \to \mathsf{Addr}$). The most important operations to perform on the heap are:

The most important operations to perform on the interpreter state are:

store expects a binding to be stored on the heap. It allocates the right-hand side on the heap and associates the object's address with the binder in the environment. The return value is the heap address of the allocated object.

readHeap looks up an address on the heap and returns the value stored at that address.

defn returns the binder an object is being bound to at its allocation given the object's address ($\texttt{defn} \in \mathsf{Addr} \rightharpoonup D$).

apply Takes the address of a closure allocated on the heap and a list of arguments to be passed to the closure and applies them to the closure.

In the following sections the different program points are presented, where the measurements are taken.

Keeping track of a variable's scope

The *external STG interpreter* evaluates expressions with the function `eval`, which matches all different types of expressions and performs the according evaluation. In the instance of a let expression, a binding is being declared by allocating a closure or thunk on the heap and associating the binder with the heap address of that object. After that, the let body is being evaluated. The pseudo code for `eval` is shown below:

$$\texttt{eval}\,(\textbf{let}\ b = r\ \textbf{in}\ e)_{\Sigma} = \begin{array}{ll} let & (addr, \Sigma') = \texttt{store}(b = r)_{\Sigma} \\ in & \texttt{eval}(e)_{\Sigma'} \end{array}$$

A newly declared binding has to be added to l with its heap address. In case b hasn't been declared before, it also has to be added to A. If the heap object is new, its access must be compared to the lexical scope of b. Thus, its address has to be added to p together with b. To calculate how much memory allocation is being caused by the binder b, the size of the allocated object has to be added to $n(b)$. The pseudo code of the modified function `eval` is shown below:

$$\texttt{eval}\,(\textbf{let}\ b = r\ \textbf{in}\ e)^{\Gamma}_{\Sigma}\ l\ j = \begin{array}{ll} let & (A^*_h, n) = \Gamma \\ & (addr, \Sigma') = \texttt{store}(b = r)_{\Sigma} \\ & l' = l \cup \{addr\} \\ & n' = n[b \mapsto n(b) + \Delta(r)] \\ & \Gamma' = (A^*_h, n') \\ in & \texttt{eval}(e)^{\Gamma'}_{\Sigma'}\ l'\ j \end{array}$$

An object's size in bytes is estimated by the operator Δ according to its structure:

$$\Delta\,(\overline{x_f}^{\,n}\,\lambda\,\overline{x_a}\,.\,e) = 8 + 8n$$
$$\Delta(K\ \overline{x}^{n}) = 8 + 8n$$

The size estimation is based on the assumption of a word size to be 8 bytes. The first word of a closure is alway the pointer to the statically allocated info table. the rest of the closures payload consists of its free variables $\overline{x_f}$, which are either pointers to other data structures or primitive values. In both cases the size of each variable is estimated to be 8 bytes. A data constructor also consists of a pointer to its info table and additionally to its arguments, which are estimated to have a size of 8 bytes.

Keeping track of an object's access

To access an object at runtime, the function `readHeap` is used which looks up the address on the heap:

$$\texttt{readHeap}(addr)_\Sigma = \begin{array}{ll} let & (\texttt{heap}, \texttt{env}) = \Sigma \\ in & \texttt{heap}(addr) \end{array}$$

To determine if the access causes an object to escape, we just check if the binder $\texttt{defn}(addr)_\Sigma$ is in scope. If it isn't, the object has escaped and we add the binder to A_h^*:

$$\texttt{readHeap}(addr)^\Gamma_\Sigma\ l\ j = \begin{array}{ll} let & (A_h^*, n) = \Gamma \\ & A_h^{*\prime} = A_h^* \cup \{\texttt{defn}(addr)_\Sigma \mid addr \notin l\} \\ & (\texttt{heap}, \texttt{env}) = \Sigma \\ in & (\texttt{heap}(addr), (A_h^{*\prime}, n)) \end{array}$$

Recovering the profiler's state at the jump to a join points

The external STG interpreter treats join point applications identically to function calls. However, to evaluate the escape analysis algorithm against the operational semantics of join points for the GHC, we have to emulate the reset of the evaluation context described in [9]. This is accomplished by capturing l for every join point definition:

$$\texttt{eval}\,(\textbf{join}\ b = r\ \textbf{in}\ e)^\Gamma_\Sigma\ l\ j = \begin{array}{ll} let & (A_h^*, n) = \Gamma \\ & (addr, \Sigma') = \texttt{store}(b = r)_\Sigma \\ & l' = l \cup \{addr\} \\ & j' = j[addr \mapsto l'] \\ & n' = n[b \mapsto n(b) + \Delta(r)] \\ & \Gamma' = (A_h^*, n') \\ in & \texttt{eval}(e)^{\Gamma'}_{\Sigma'}\ l'\ j' \end{array}$$

Before performing a call to a function f, we check if f is actually a join point by looking up its address $addr$ in j. If f is a join point, we recover l from the join

point's definition by setting it to $j(addr)$:

$$\begin{aligned}
&\texttt{eval}\,(f\ \overline{a})^{\Gamma}_{\Sigma}\ l\ j = \\
&\quad \textit{let} \quad (A^*_h, n) = \Gamma \\
&\qquad\quad (\texttt{heap}, \texttt{env}) = \Sigma \\
&\qquad\quad addr = \texttt{env}(f) \\
&\qquad\quad l' = \begin{cases} j(addr) & \text{if } j(addr) \neq \bot \\ l & \text{else} \end{cases} \\
&\quad \textit{in} \quad \texttt{apply}(addr, \overline{a})^{\Gamma}_{\Sigma}\ l'\ j
\end{aligned}$$

4.4. Results

112 benchmarks have been run from the nofib test sets *imaginary*, *spectral*, *real* and *shootout*. All benchmarks have been run in fast mode. Since the modified interpreter consumes a lot of memory, 9 of them have crashed due to not enough main memory on a machine with 32GB of RAM. One benchmark had to be aborted due to too long computation time. The raw data produced from running the benchmarks is provided in appendix A.2.

4.4.1. Soundness of the escape analysis algorithm

As explained in section 4.1, one way to evaluate the soundness of the algorithm is, to identify the false negatives F_s, which is the set of binders that are wrongfully assumed to not escape during runtime. For an escape analysis to provide a correct result, we expect F_s to be empty. During all benchmarks that have been performed on the final implementation of the escape analysis as part of this thesis, no false positives have been identified. This favors the hypothesis, that the escape analysis algorithm is correct.

4.4.2. Effectiveness of the escape analysis algorithm

Figure 4.1 shows the relation of two different metrics: It displays the portion of binders that could have been allocated on the stack $\left(\frac{|A_s|}{|A|}\right)$ at the x axis and the portion of stack allocated memory (S) for each benchmark run at the y axis.

The fact that these two metrics do not correlate, confirms the assumption that the portion of non-escaping binders isn't representative of the effectiveness. Surprisingly, the vast majority of binders in the benchmark programs can be allocated on the stack (always $> 96\%$). However, in most runs, the few binders that had to be allocated on the heap were responsible for the majority of the allocated memory.

Figure 4.2 compares the portion of stack allocated memory (S) based on the escape analysis with the upper bound (S^*) for each benchmark run. For each run, the value of S and S^* is visualized by the with of the colored rectangles, which are stacked vertically. The height of the areas are proportional to the total amounts of allocated memory for each benchmark run. The portion of blue area represents the value of S

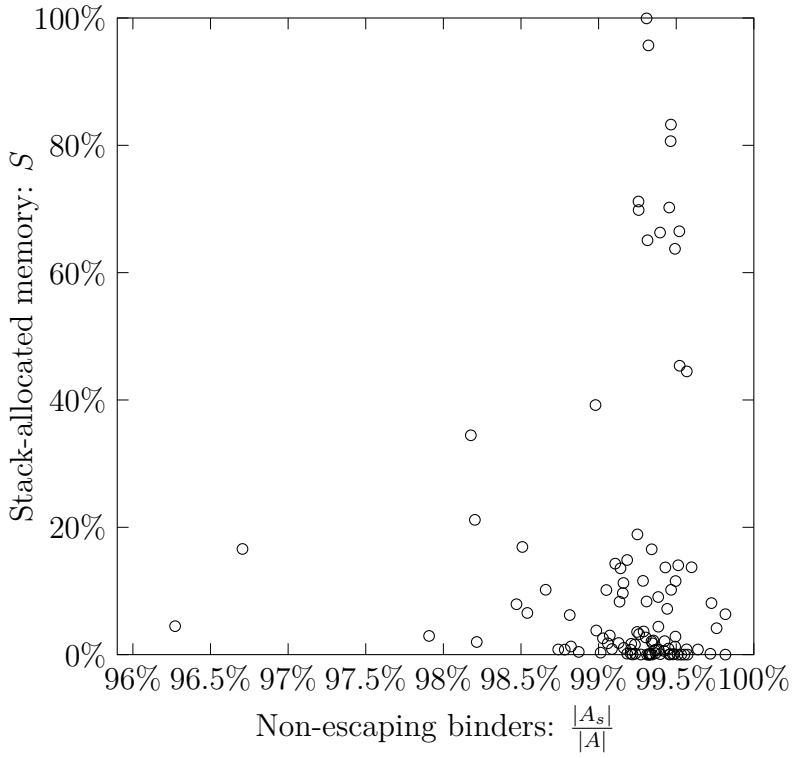


Figure 4.1.: Portion of stack-allocated memory in relation to the portion of non-escaping binders

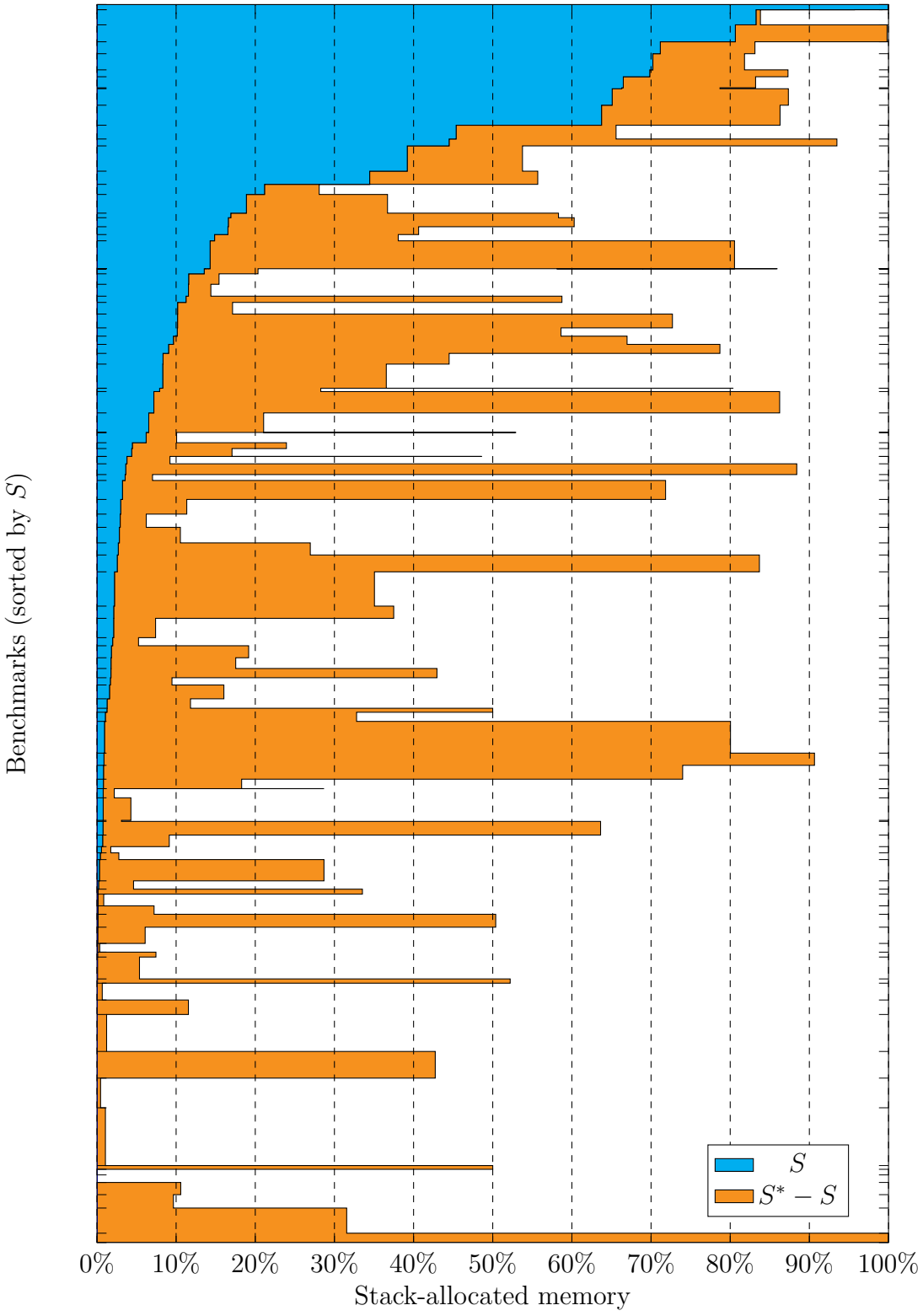


Figure 4.2.: Portion of stack allocated memory compared to the upper bound

for each benchmark respectively and also in total. The blue area plus the orange represents S^*.

Of the total amount of allocated memory, the escape analysis has identified ca. 13.7% of allocated memory that could have been allocated on the stack. The top ten of the benchmarks are responsible for over 80% of the total stack-allocated memory. Apparently, there is a small portion of programs that benefit from this specific escape analysis. The orange area shows a few spikes, which indicates, that S^* doesn't have a strong correlation with S This could indicate, that there are still missed opportunities for bindings to be allocated on the stack. It could, however, also be due to a lack of path coverage as mentioned in section 4.1.

These spikes need to be investigated more deeply because for some benchmarks where the escape analysis didn't have a significant impact, they shows a large amount of missed opportunities.

The following two scenarios are possible explanations for this phenomenon:

- A function is never called in a context, where its return value is scrutinized entirely. A binding defined in that function might be returned, but none of the callers are demanding the value of that binding. The algorithm wouldn't be able to detect this because it is context insensitive.
- A closure is being used as an argument in a primitive operation performing a side effect. For example, the address of a closure is being written into a mutable variable that is defined outside the lexical scope. However, the escape might not be detected at runtime when the mutable variable is never being read outside the lexical scope.

To identify missed opportunities for some benchmarks that are causing these spikes, we look at the elements in F_h.

The biggest spike is caused by the benchmark program `S`. It is also a good example for the first scenario:

```
n :: Int
n = 10000000

times :: Monad m => Int -> m a -> m ()
times n ma = go n where
    go 0 = pure ()
    go n = ma >> go (n - 1)
{-# inline times #-}

main = do

    putStrLn "S"
    evaluate $ S.runState (times n $ (S.modify (+1))) 0
```

The program initializes a state monad with `0` and increments it by `1` repeatedly. By analysing the STG code, it becomes apparent that most allocations are being performed in the function `$wgo`:

```
ds_r2SI :: GHC.Integer.Type.Integer
src<<no location info>>
[GblId, Unf=OtherCon []] =
    CCS_DONT_CARE GHC.Integer.Type.S#! [1#];

Rec {
$wgo [InlPrag=NOUSERINLINE[2], Occ=LoopBreaker]
    :: GHC.Prim.Int#
        -> GHC.Integer.Type.Integer -> (# (), GHC.Integer.Type.Integer #)
src<<no location info>>
[GblId, Arity=2, Str=<S,1*U><L,U>, Unf=OtherCon []] =
    \r [ww_s2SO w_s2SP]
        case ww_s2SO of wild_s2SQ [Occ=Once] {
            __DEFAULT ->
                let {
                sat_s2SS [Occ=Once] :: GHC.Integer.Type.Integer
                src<<no location info>>
                [LclId] =
                    \u [] GHC.Integer.Type.plusInteger w_s2SP ds_r2SI;
                } in
                case -# [wild_s2SQ 1#] of sat_s2SR [Occ=Once] {
                __DEFAULT -> $wgo sat_s2SR sat_s2SS;
                };
            0# -> (#,#) [GHC.Tuple.() w_s2SP];
        };
end Rec }
```

The binding `sat_s2SS` represents the updated state, which is the previous state incremented by `1`. When running the benchmark, this binding is responsible for nearly 100% of heap allocations. However, the state is represented as a pair and the function `evaluate` evaluates the final state just to *weak head normal form*, it doesn't demand the fields of that pair. Having this knowledge would enable us to allocate `sat_s2SS` on the stack.

Note, that allocating `sat_s2SS` on the stack would only be sound, if all callsites of `$wgo` are known and the final state is never evaluated beyond weak head normal form.

The benchmark `reverse-complement` shows an example for the second scenario. The binding `sat_sa8M` from the package `GHC.IO.Handle.Text` is responsible for ca. 31% of memory allocations:

```
let {
    sat_sa8M [Occ=Once] :: GHC.IO.Buffer.Buffer GHC.Word.Word8
    src<<no location info>>
    [LclId] =
        CCCS GHC.IO.Buffer.Buffer! [dt4_sa87
                                    dt5_sa88
                                    ds9_sa89
                                    dt6_sa8a
                                    sat_sa8L
                                    dt8_sa8c];
  } in
    case
        writeMutVar# [dt_sa7V sat_sa8M GHC.Prim.void#]
    of
    s2#_sa8N [Occ=Once]
    {
    (##) ->
    case +# [ipv_sa8e ipv1_sa8g] of sat_sa8O [Occ=Once] {
    __DEFAULT ->
    let {
      sat_sa8P [Occ=Once] :: GHC.Types.Int
```

```
      src<<no location info>>
      [LclId] =
          CCCS GHC.Types.I#! [sat_sa8O];
    } in  Unit# [sat_sa8P];
    };
  };
```

Despite being stored in a mutable variable, that is defined outside the binding's lexical scope, the closure for `sat_sa8M` appears to never be accessed outside the lexical scope during a benchmark run.

5. Conclusion and future work

The results from the evaluation show that in the context of a lazy language like Haskell, an escape analysis can have a significant impact on compile time optimization for some programs. The escape analysis algorithm developed as part of this thesis produces sound results for every benchmark that has been run during the evaluation. The data collected from the benchmark runs support the assumption that, how much heap allocation can be shifted to the stack, depends on a relatively small subset of bindings. The custom profiling functionality for the external STG interpreter turns out to be a useful tool to identify missed opportunities of an escape analysis and to provide direction for the improvement of escape analysis algorithms.

Based on the findings of this thesis, further studies can delve into the following research topics:

- Evaluating the escape analysis algorithm for the GHC using other, more practical programs
- Implementing compile time optimizations for stack allocation and closure reuse for the GHC
- Investigating speedup, memory consumption and space safety for different compile time optimizations based on escape analysis for the GHC
- Extending the escape analysis algorithm for the GHC to be context-sensitive.

Bibliography

[1] B. Blanchet, "Escape analysis for java^{tm}: Theory and practice," *ACM Trans. Program. Lang. Syst.*, vol. 25, p. 713–775, nov 2003.

[2] T. Kotzmann and H. Mössenböck, "Run-time support for optimizations based on escape analysis," in *In Proceedings of the International Symposium on Code Generation and Optimization*, pp. 49–60, IEEE Computer Society, 2007.

[3] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones, "Parallel generational-copying garbage collection with a block-structured heap," in *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, (New York, NY, USA), p. 11–20, Association for Computing Machinery, 2008.

[4] S. Carr, K. S. McKinley, and C.-W. Tseng, "Compiler optimizations for improving data locality," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, (New York, NY, USA), p. 252–262, Association for Computing Machinery, 1994.

[5] Z. Shao and A. W. Appel, "Efficient and safe-for-space closure conversion," *ACM Trans. Program. Lang. Syst.*, vol. 22, p. 129–161, jan 2000.

[6] B. Blanchet, "Escape analysis: Correctness proof, implementation and experimental results," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, (New York, NY, USA), p. 25–37, Association for Computing Machinery, 1998.

[7] S. Peyton Jones, "Implementing lazy functional languages on stock hardware: The spineless tagless g-machine," *Journal of Functional Programming*, vol. 2, pp. 127–202, July 1992.

[8] S. Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice Hall, January 1987.

[9] L. Maurer, Z. Ariola, P. Downen, and S. Peyton Jones, "Compiling without continuations," in *ACM Conference on Programming Languages Design and Implementation (PLDI'17)*, pp. 482–494, ACM, June 2017.

[10] "ticky ticky." `https://gitlab.haskell.org/ghc/ghc/-/wikis/debugging/ticky-ticky`, 2022. Last accessed 18 May 2022.

[11] C. Hruska, "Ghc whole program compiler project." `https://github.com/grin-compiler/ghc-whole-program-compiler-project`, 2022. Last accessed 2 May 2022.

[12] "nofib." `https://gitlab.haskell.org/ghc/nofib`, 2022. Last accessed 18 May 2022.

Erklärung

Hiermit erkläre ich, Sebastian Scheper, dass ich die vorliegende Masterarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

Acknowledgements

Thanks to Sebastian Graf, Andreas Klebinger and Simon Peyton Jones for initiating the idea of implementing an escape analysis for the GHC and discussing possible optimizations concerning stack allocation. In particular, I want to thank Sebastian Graf who also supervised this master thesis and supported me in my work tremendously. I also want to thank Csaba Hruska for providing an introduction to his external STG interpreter.

A. Appendix

A.1. Computation of metrics

The following information can be derived from A_h^* and n after the profiling of the benchmark P:

A: Binders allocated during the execution of P

$$A = \{b \in D \mid n(b) > 0\}$$

A_s: Binders that could have allocated their objects on stack during the execution of P according to the escape analysis

$$A_s = A \setminus B$$

A_h: Binders that couldn't have allocated their objects on stack during the execution of P according to the escape analysis

$$A_h = A \cap B$$

A_s^*: Binders allocated during the execution of P whose objects never got accessed outside their lexical scope

$$A_s^* = A \setminus A_h^*$$

F_h: Binders that couldn't have allocated their objects on stack according to the escape analysis but actually never got accessed outside their lexical scope during the execution of P (false positives)

$$F_h = A_s^* \setminus A_s$$

F_s: Binders that are wrongfully assumed to not escape during the execution of P (false negatives)

$$F_s = A_h^* \setminus A_h$$

This is the essential value indicating the soundness of the escape analysis. For the result of an escape analysis to be sound, we expect $F_s = \emptyset$ to be true.

M_s: Amount of stack-allocated memory caused by let-bound objects during the execution of P

$$M_s = \sum_{b \in A_s} n(b)$$

M_h: Amount of heap-allocated memory caused by let-bound objects during the execution of P

$$M_h = \sum_{b \in A_h} n(b)$$

M_s^*: Upper bound for the amount of stack-allocated memory caused by let-bound objects during the execution of P

$$M_s^* = \sum_{b \in A_s^*} n(b)$$

M_h^*: Lower bound for the amount of heap-allocated memory caused by let-bound objects during the execution of P

$$M_h^* = \sum_{b \in A_h^*} n(b)$$

S: Portion of stack-allocated memory caused by let-bound objects during the execution of P

$$S = \frac{M_s}{M_s + M_h}$$

S^*: Upper bound for the portion of stack-allocated memory caused by let-bound objects during the execution of P

$$S^* = \frac{M_s^*}{M_s^* + M_h^*}$$

A.2. Raw data of the Evaluation

P	$\lvert A_s \rvert$	$\lvert A_h \rvert$	$\lvert A_s^* \rvert$	$\lvert A_h^* \rvert$	$\lvert F_h \rvert$	$\lvert F_s \rvert$	M_s	M_h	M_s^*	M_h^*
n-body	44661	311	44866	106	205	0	44008640	27416	44029304	6752
spectral-norm	44604	305	44809	100	205	0	517952	23296	536880	4368
power	44325	238	44485	78	160	0	107297080	21573776	108015920	20854936
digits-of-e1	44194	238	44366	66	172	0	116827536	28026936	144591944	262528
exact-reals	44467	333	44683	117	216	0	72537200	29385704	84722408	17200496
digits-of-e2	44201	242	44368	75	167	0	96938496	41094560	112921856	25111200
primetest	44225	330	44438	117	213	0	41163992	17778864	51456880	7485976
rsa	44261	214	44391	84	130	0	62700488	31592272	78458320	15834440
pidigits	44201	269	44395	75	194	0	5418112	2754504	6432472	1740144
bernouilli	44235	305	44430	110	195	0	91625800	49145080	122963264	17807616
gcd	44185	226	44345	66	160	0	109028800	62012592	147605088	23436304
integer	44190	212	44326	76	136	0	53382168	64259296	77148616	40492848
mandel2	44217	192	44357	52	140	0	26243080	32747960	55145656	3845384

P	$\|A_s\|$	$\|A_h\|$	$\|A_s^*\|$	$\|A_h^*\|$	$\|F_h\|$	$\|F_s\|$	M_s	M_h	M_s^*	M_h^*
gamteb	44657	460	44883	234	226	0	84712184	131379968	116163216	99928936
scs	44861	833	45338	356	477	0	38799872	73815032	62712104	49902800
transform	44685	818	45015	488	330	0	18311944	68184624	24283368	62213200
awards	44248	335	44516	67	268	0	30160640	129579168	58635672	101104136
mate	44531	674	44924	281	393	0	6578496	32327192	22683904	16221784
fluid	45125	1537	45839	823	714	0	12832048	64489416	46617344	30704120
boyer2	44552	295	44728	119	176	0	11046312	55722640	27132536	39636416
nucleic2	48651	400	48863	188	212	0	7944216	45514208	20348104	33110320
dom-lt	52495	473	52799	169	304	0	34088200	204400024	192089112	46399112
tak	44169	216	44329	56	160	0	1392	8528	6840	3080
kahan	45359	183	45488	54	129	0	752	4728	3184	2296
rfib	44186	254	44370	70	184	0	2856	18000	17920	2936
symalg	44699	387	44941	145	242	0	5915096	37730288	8879840	34765544
eliza	44614	321	44816	119	202	0	10371688	79154176	13807464	75718400
mandel	48771	247	48917	101	146	0	11722472	89726696	14598528	86850640
sphere	44435	377	44673	139	238	0	6003296	47353760	31342656	22014400
gg	45916	624	46178	362	262	0	10134096	89353696	17041360	82446432
puzzle	44288	237	44457	68	169	0	12022736	106082128	85863368	32241496
rewrite	44419	426	44683	162	264	0	7177272	63572336	41475800	29273808
parser	45054	383	45291	146	237	0	6796320	63550648	47110488	23236480
maillist	44234	274	44442	66	208	0	6967360	70023720	60591216	16399864
wave4main	44259	308	44450	117	191	0	7648736	83994312	40771696	50871352
para	44332	387	44574	145	242	0	17124568	188802456	75273600	130653424
hidden	44563	122	44657	28	94	0	41704	473592	414016	101280
simple	44631	693	45018	306	387	0	2236824	26014848	7987896	20263776
minimax	45218	254	45365	107	147	0	13119256	169401704	157422712	25098248
fem	44543	659	44953	249	410	0	10931768	156115544	35174528	131872784
VSD	44104	81	44164	21	60	0	192	2832	1600	1424
linear	44769	537	45008	298	239	0	5454048	82311232	8834520	78930760
anna	47801	1851	48372	1280	571	0	2226888	47582464	11922560	37886792
event	44308	274	44478	104	170	0	2958512	64446592	11500304	55904800
scc	44145	106	44220	31	75	0	200	4624	2344	2480
expert	45260	464	45532	192	272	0	2385088	60579000	5807256	57156832
cichelli	44334	317	44560	91	226	0	3343888	88556840	81245784	10654944
fft2	48772	370	48958	184	186	0	1794872	48620816	3538296	46877392
paraffins	44222	329	44446	105	224	0	5232744	156936368	116494504	45674608
circsim	44426	416	44659	183	233	0	3765256	121725224	14239144	111251336
hpg	44932	960	45167	725	235	0	3314408	109600504	7031776	105883136
cse	44294	225	44430	89	136	0	3760672	129124760	14003024	118882408
wang	44251	310	44426	135	175	0	2788096	100690608	27889056	75589648
pic	44343	435	44623	155	280	0	3693360	140395800	120584552	23504608
fibheaps	44270	288	44441	117	171	0	6489680	286455160	102693736	190251104
fft	48767	282	48963	86	196	0	2214824	102323416	39192104	65346136
cryptarithm2	44270	294	44456	108	186	0	3485840	161700960	12234992	152951808
veritas	48471	881	48776	576	305	0	1370392	67726088	3628256	65468224
comp_lab_zift	44455	391	44672	174	217	0	1872104	100734728	19661864	82944968
prolog	44385	422	44586	221	201	0	1629568	90184288	16095496	75718360
atom	44229	291	44434	86	205	0	1396112	77602096	33933392	45064816
typecheck	44535	355	44748	142	213	0	1048712	60820448	5864032	56005128
ida	44318	342	44545	115	227	0	1873536	116285232	18947032	99211736
infer	44621	532	44858	295	237	0	1033944	79742048	9544184	71231808
fasta	49309	251	49474	86	165	0	431080	33719480	17074776	17075784
sorting	44280	374	44521	133	241	0	826296	77305552	25626640	52505208
treejoin	44227	244	44401	70	174	0	2588912	268639480	217082848	54145544
reptile	44909	414	45187	136	278	0	920592	103435032	94594856	9760768
banner	44535	194	44669	60	134	0	978816	117403496	87594232	30788080
cacheprof	46898	599	47270	227	372	0	657856	80372096	14811528	66218424
pretty	44266	160	44370	56	104	0	216	26688	7704	19200
lift	45460	560	45637	383	177	0	615040	77641312	1712176	76544176
lcss	44185	278	44382	81	197	0	1512336	190956800	8251176	184217960
wheel-sieve1	44194	273	44358	109	164	0	70504	9048160	282664	8836000
solid	44428	355	44647	136	219	0	891928	115875784	74293896	42473816
mkhprog	44247	284	44404	127	157	0	712168	98142632	9031096	89823704
lambda	45544	265	45727	82	183	0	304896	52912424	929056	52288264
bspt	46240	527	46463	304	223	0	244608	57219680	1588456	55875832
fulsom	45035	449	45291	193	256	0	568320	181190208	52128224	129630304
life	44219	282	44393	108	174	0	150096	71042608	3291056	67901648
parstof	45458	374	45705	127	247	0	64096	42305120	14209056	28160160
genfft	44265	354	44428	191	163	0	148904	100136576	857864	99427616
multiplier	44357	342	44595	104	238	0	100488	71780696	5185664	66695520
reverse-complement	44137	124	44217	44	80	0	149672	110191408	55585368	54755712
calendar	44257	301	44425	133	168	0	136680	138731976	8455552	130413104
primes	44175	228	44337	66	162	0	60904	74447360	262664	74245600
clausify	44242	268	44424	86	182	0	33336	41792736	3116832	38709240
sched	44360	297	44533	124	173	0	122232	186846776	10046704	176922304
knights	44376	240	44526	90	150	0	21696	36583120	19109056	17495760
boyer	44837	235	45006	66	169	0	47304	143562656	961864	142648096
grep	45185	356	45417	124	232	0	25240	123759416	14302656	109482000
compress2	44224	242	44345	121	121	0	56472	315923400	3859256	312120616
ansi	44308	325	44544	89	236	0	37720	227834696	97388264	130484152
binary-trees	44502	293	44721	74	219	0	28896	253558336	1160232	252427000
compress	44695	192	44804	83	109	0	19504	493907680	5278272	488648912
x2n1	48687	219	48846	60	159	0	896	32005848	16004152	16002592
queens	44172	218	44331	59	159	0	632	47838088	3616	47835104
exp3_8	44210	208	44365	53	155	0	744	64535792	3632	64532904
listcompr	44412	301	44573	140	161	0	736	104326880	11043856	93283760
listcopy	44413	302	44574	141	161	0	736	114342880	11043856	103299760
fish	44437	308	44582	163	145	0	528	215031488	67843584	147188432
S	44106	81	44166	21	60	0	192	80002768	80001584	1376

nofib test sets

- imaginary
- spectral
- real
- shootout

Crashed due to not enough memory

- gen_regexps
- constraints
- cryptarithm1
- last-piece
- CS
- FS
- CSD
- fannkuch-redux
- k-nucleotide

Aborted due to too long computation time

- VS