Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Matemáticas e Informática

Trabajo Fin de Grado

Análisis de Coste en Programas Funcionales Usando CiaoPP

Autor: David Munuera Mazarro

Tutor: Manuel Hermenegildo Salinas

Madrid, Junio de 2020

Trabajo Fin de Grado
Grado en Matemáticas e Informática
Título: Análisis de Coste en Programas Funcionales Usando CiaoPP

Junio de 2020

Autor: David Munuera Mazarro
Tutor: Manuel Hermenegildo Salinas
ETSI Informáticos
Universidad Politécnica de Madrid

Abstract

Estimating and measuring resource and cost usage of programs is a key research area in software development, as that allows developed software to be more cost-effective, more usable, and more reliable. There are several approaches to this, like profiling, which require you to run the program you want to analyze in order to measure resource usage. This is not always feasible depending on the scale and character of the software, and even when it is doable, what you get are estimations that can be more or less accurate depending on the quality of the tool used for performing said analysis.

Static analysis tries to address this by making guarantees about a program's behavior prior to its execution; however, since this often requires added annotations or some sort of aid coming from the programmer, they can be impractical to work with. Ideally, we are striving for static, fully automatic ways of reasoning about software and its resource usage. While there exist tools for doing static resource analysis over Haskell programs, none of them are fully automatic.

This thesis proposes and develops ***hs-to-ciao***, a tool in the form of a plugin for GHC (the industry standard Haskell compiler) for translating Haskell programs into Ciao Prolog source code in order to take advantage of the already existing resource analysis framework within Ciao's preprocessor, CiaoPP. The tool serves a double purpose: on the one hand, it provides a fully automatic and static resource analysis solution for Haskell; on the other hand, it can be used as a general purpose source-to-source compiler from Haskell programs into Ciao programs that use the functional syntax provided by the language as an extension.

Abstract (in Spanish)

La estimación y medición del uso de recursos y costes de los programas es un área clave en la investigación del desarrollo de *software*, ya que permite al *software* desarrollado ser más económico, usable y fiable. Hay varios enfoques dentro de este área, como el *profiling*, que requiere de la ejecución del programa a analizar para poder medir el uso de recursos. Esto no siempre es posible dependiendo de la escala y naturaleza del *software*, y en los casos en los que se puede, las estimaciones serán más o menos acertadas dependiendo de la calidad de la herramienta usada para dicho análisis.

El análisis estático intenta solventar este problema ofreciendo garantías acerca del comportamiento de un programa antes de su ejecución; sin embargo, a menudo es necesario anotaciones añadidas sobre el programa u otro tipo de asistencia por parte del programador, lo cual puede llegar a ser impráctico. Idealmente, se persigue llegar a métodos de análisis que sean estáticos y completamente automáticos para poder razonar acerca del *software* y su uso de recursos. Mientras que existen herramientas para realizar análisis estático sobre programas en escritos en Haskell, ninguna es completamente automática.

Este trabajo propone y desarrolla ***hs-to-ciao***, una herramienta en forma de plugin para GHC (el compilador estándar de Haskell) para traducir programas en Haskell a código fuente en Ciao Prolog con el fin de aprovechar las posibilidades que ofrece CiaoPP, el preprocesador de Ciao, que ofrece un *framework* para análisis de recursos. La herramienta cumple un doble propósito: por un lado, ofrece una solución completamente automática para análisis estático de recursos para Haskell; por otro, se puede usar como transpilador de propósito general para traducir programas escritos en Haskell a programas escritos en Ciao que usan la sintaxis funcional ofrecida por el lenguaje a modo de extensión.

Acknowledgements

First of all, I want to thank Niki Vazou for being my advisor at IMDEA Software and providing help at all times, clearing up a lot of technical questions and ensuring that the project stayed on track, everything in the nicest and kindest way possible.

I also want to thank everyone that has provided me with assistance throughout the course of this project: Manuel H., Pedro, José, Maximiliano, and the Ciao Development Team as a whole.

Big thanks to all of the good teachers I've had up to this point: not every teacher has been a good teacher, but every teacher that has been good deserves my sincerest gratitude.

Finally, I want to thank my family, an specially my mom, dad, and Santi: for being a constant support for everything I've wanted to do during my life, including my journey through this bachelor's degree; my friend Gerard, for being by my side so many years (and more to come!), and my flatmate Óscar, which now is almost like a brother to me.

Agradecimientos

Primero, quiero dar las gracias a Niki Vazou por ser mi tutora durante mi estancia en IMDEA Software y haberme ayudado en todo momento, despejando muchas dudas técnicas y asegurándose de que el proyecto mantuviera su curso, todo ello de la forma más agradable y amable posible.

También quiero dar las gracias a todos los que me han asistido a lo largo del desarrollo del proyecto: Manuel H., Pedro, José, Maximiliano, y el equipo de desarrollo de Ciao en su totalidad.

Muchas gracias a todos los profesores buenos que he tenido hasta este momento: no todo profesor ha sido buen profesor, pero todo buen profesor merece mi más sincera gratitud.

Finalmente, quiero agradecer a mi familia, y especialmente a mi madre, mi padre, y Santi: por haber sido un apoyo constante para todo lo que he querido hacer a lo largo de mi vida, incluyendo mi paso por este grado y la universidad; a mi amigo Gerard, por estar a mi lado tantos años (¡y los que quedan!), y a mi compañero de piso Óscar, que ahora es prácticamente un hermano.

Table of contents draft

Chapter 1

Introduction

1.1 Predicting software costs

When talking about computer programs, “cost” is a very broad encompassing term that can refer to things such as:

- Energy consumption
- CPU utilization
- Execution time
- Computational complexity (lower and upper bounds, big-O...)
- Space complexity
- Number of calls to a procedure
- Granularity in parallelism

Among a large number of other examples.

These metrics can be invaluable, allowing you to know which algorithm fits your needs better in a particular situation: minimizing battery or memory usage in embedded devices, achieving cost-effective solutions in enterprise software so you get the most returns, or, by combining the metrics, deciding whether you have to worry or not about a function’s computational cost/complexity by knowing the number of times it’s called (if it’s big, then you may want to tweak it, and if it’s small perhaps you’re better off spending your time on the rest of your code).

In an effort to try to make things easier for anyone to get a hold of these metrics for, ideally, most programs in most programming languages, studies in the field of *automatic cost analysis* arose.

1.2 What is automatic cost analysis?

Automatic cost analysis is, as the name suggests, a series of techniques oriented towards getting results and measurements for these costs, and in particular interest, without having to run the program several (or even many) times to be able to perform an in hindsight statistical analysis, as very often these repeated tests can take quite a long time or cost a lot of money.

There are two main aspects of cost analysis: *static cost analysis*, which tries to get a measure of costs at compile-time or before a program is executed, and *run-time cost analysis*, which requires the program to be (partially or completely) run at least one time to extract conclusions about its cost properties. The advantage that static analysis provides is that you don't have to run the code to get cost information, but many times it is harder (if not impossible) to get certain information without having run the program. On the other hand, run-time analysis helps easing this issue. A mixed approach is often used [3, 14] in order to get the best of both worlds, using the information that static analysis provides to help in the run-time analysis.

Having said that, we would also like to have the most general way to tackle the problem; and in an effort to do so, both existing techniques like abstract interpretation [1] as well as new, developing techniques for this particular field [13] have been used.

We are also interested not only in having techniques for certain types of costs, but also in having a way to make our techniques parametric with respect to the used resources. As such, [15] proposes the following definition for *resource*:

> "A *resource* is a user-defined notion which associates a basic cost function with elementary operations in the base language and/or to some predicates in libraries. In this sense, each resource is essentially a user-defined *counter*. The user gives a name (e.g. `bits_received`) to the counter and then defines via assertions how each elementary operation in the program increments or decrements that counter."

It also provides a general framework for performing user-definable resources cost analysis for logic programs with this definition in mind. We are particularly interested in this for the purposes of this work.

1.3 The problem

The biggest issue with cost analysis, as with any theoretical tool with a sufficiently large scope, is that almost every programming language requires its own implementation, as each one has its own set of semantics and level of abstraction; e.g. it's quite easier to get a hold on the execution time a given program will have if we analyze the assembly code of said program than its higher-level original source code, and it doesn't make much sense to analyze the number of β-reductions a function application will require in a functional programming language as it's an inherently higher-level concept whose meaning gets lost once your program is compiled.

Haskell, one of the most widely used functional programming languages, doesn't have any sort of tool to do this in a fully automatic way that we know of as of now. As stated previously, having some kind of way to perform cost analysis over programs can prove to be quite useful, and given that a fully automatic approach is the most comfortable way to do so from the developer's perspective, it makes sense to try and tackle this problem.

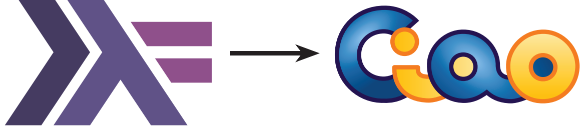
1.3.1 How can we approach the problem?

One way to deal with the problem would be to create a tool from scratch that targets Haskell, and build all of the analysis utilities from there. However, given the similarities between *logic programming* and *functional programming* (both fall under the hood of *declarative programming*), we thought an interesting way to try to approach the problem would be to create a source-to-source compiler that takes Haskell source code and translates it into **Ciao** source code (a logic programming language sharing many similarities with Prolog), to thereafter perform our resource analysis over the Ciao code with the **CiaoPP** preprocessor and get our conclusions back for our original Haskell code. Of course, for this to yield any valid results, the program you get from the translation process should present an equivalent behavior to the original program it came from, and that could pose a few limitations into which things can be translated given a couple fundamental differences between the two languages, like the differing evaluation strategies (Haskell having *lazy evaluation* and Ciao *strict evaluation*).

1.3.2 Why CiaoPP?

The *CiaoPP* preprocessor has a well-developed toolkit for performing said cost analysis tasks; it even provides a GUI for doing so. Besides that, it is parametric with regards to user-definable resources, a trait that is highly desirable in these sort of tools. The advantage that gives us is that we don't have to build all of the analysis tools for Haskell, since they already exist within the Ciao ecosystem.

1.4 What is *hs-to-ciao*?



hs-to-ciao's "logo"

The tool we propose is called *hs-to-ciao*, and its goal is to offer an alternative as well as improving upon what's already out there in the static resource analysis and automatic resource analysis fields applied to Haskell.

It consists of a **plugin** for the industry standard Haskell compiler, the ***Glasgow Haskell Compiler*** (shorthand **GHC**), which performs the two following main tasks:

1. Translate some Haskell source code into Ciao source code
2. Perform an automatic analysis over the translated Ciao source code

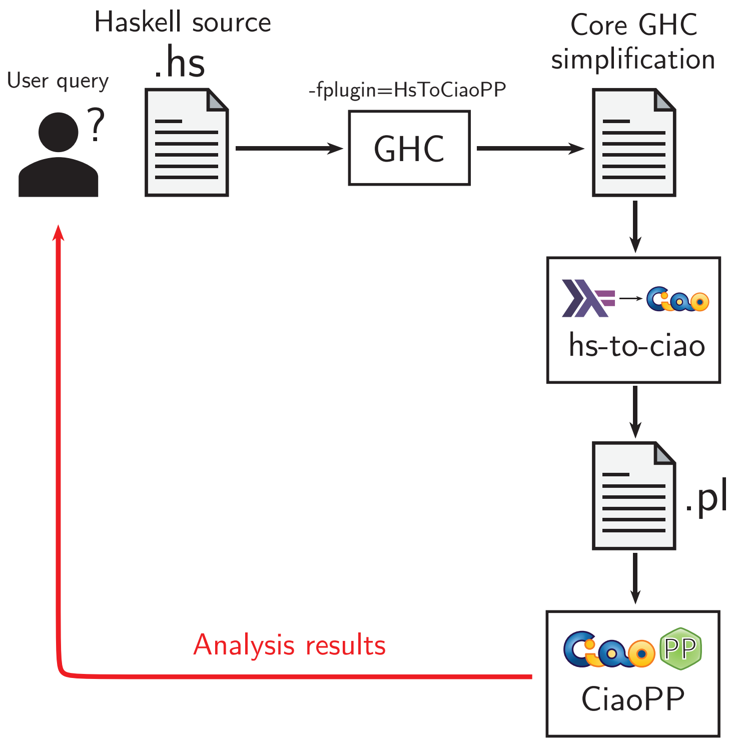


Figure 1.1: Overview of *hs-to-ciao*'s usage pipeline

The general workflow pipeline with the plugin is as shown in figure 1.1. First, the user wants to ask the plugin to perform a specific kind of analysis from a list supported by the plugin. For that, they have to first ask GHC to compile the Haskell source files with the flag `-fplugin=HsToCiaoPP`. Then, the plugin will prompt the user to select the kind of analysis they want, and after that the Haskell source gets converted down to `Core`, one of the intermediate languages that GHC uses. From there, the Core representation of the functions in the source file gets converted into a Ciao source code file with some added annotations for CiaoPP to be able to do its job; that translation is added into a `.pl` file into the plugin's output folder. Finally, a script written in Ciao gets executed, with all the info CiaoPP needs to know to perform whatever kind of analysis (i.e. which flags need to be activated), and the script calls CiaoPP. The results from the analysis are arranged nicely for the user in the form of a `.txt` located in the plugin's output folder.

1.4.1 Installation and usage instructions

For instructions on installation and usage of the plugin, please refer to the instructions found in the GitHub public repository of the plugin:

`github.com/imdea-software/hs-to-ciao`

The plugin is currently being tested with Travis CI support on the GitHub repository for all stable versions of GHC from `8.6.5` up to `8.10.1` ☺

Chapter 2

Development

Before describing what the core parts of the plugin are, and how they work and interact, it should be noted that two terms that are used throughout the rest of the thesis, *functions* and *predicates*, carry some difference in the nuances with which they're used; *functions* can refer to either Haskell's functions, whose context should be clear, or Ciao's predicates in functional syntax, which **still are** predicates under the hood once you remove the syntactic sugar that comes with the functional syntax modules for Ciao, but are treated as a basic construction in the translation, since the key to the approach has been to mimic Haskell programs' behavior in the Ciao translated files.

2.1 *hs-to-ciao* as a GHC plugin

The plugin's internal workings can be separated in the following core parts:

- Main (see `HsToCiaoPP.hs`): the topmost functions in the plugin, and the actual part that gets the required info from GHC directly, mostly the Core bindings for the program that is being compiled and is about to be translated.
- CiaoSyn (see `CiaoSyn.hs`): our representation of Ciao's syntax, but slightly altered since it's missing a few things which I deemed were dispensable for the purposes of the plugin -i.e. infix operators and operator precedence-, and in turn it was expanded to support Ciao's functional syntax from its `functional` and `hiord` modules.
- Translation (see `MainTranslation.hs`, `DataTypesTranslation.hs`, `Environment.hs`, `CoreInstances.hs` in `Translation`): this is the central part in the plugin, and is in charge of taking the Core bindings provided by `HsToCiaoPP.hs` of the functions about to be translated, and transforms them into a Ciao function encoded as a CiaoSyn term; after that, all the translated terms are outputted to a file as the translated program.
- Pretty printing (see `GeneralPrinter.hs`, `AnalysisKinds.hs` and `BigO.hs` in `PrettyPrinters`): this provides support for custom pretty printers for each of the analysis kinds (e.g. Big-O) that we would want to add, with relatively easy extensibility; more on that in subsection 3.3.2.
- `ciao_prelude` (see `IDDictionary.hs` and `lib/ciao_prelude.pl`): this provides basic translations to some of the functions that can be found in Haskell's `Prelude`, so that the user doesn't have to add them to the source about to be translated and avoiding po-

tential issues (section 3.2) involving type polymorphism of the original functions lying in `Prelude`. This also supports extensibility; more on that in subsection 3.3.3.

Besides those, theres an extra component, the **Embedder** (see `Embedder.hs`), which is responsible for adding any functions that the target files require from `ciao_prelude` to the target files themselves *after* the translation but `before` any kind of analysis is performed. This is because as I'm writing these lines, CiaoPP doesn't support importing functions/predicates from other modules, so you need to have every predicate you want to analyze in the same Ciao source file. In the future, this Embedder should disappear in favor of just adding a `:- use_module(...)` clause or similar at the beginning of the translated source about to be analyzed, if CiaoPP gets to the point of supporting this.

2.1.1 Translating from Core to Ciao

As stated previously, we have defined what we call **CiaoSyn**, an encoding of a variation on Ciao's functional syntax within a Haskell **algebraic data type** definition.

```
newtype CiaoRegtype = CiaoRegtype (CiaoId, [(CiaoId, [CiaoId])])

newtype CiaoProgram = CiaoProgram [CiaoFunctor]

data CiaoFunctor = CiaoFunctor
  { functorName :: CiaoFunctorName,
    functorArity :: Int,
    functorHsType :: ([Type], Type),
    functorMetaPred :: CiaoMetaPred,
    functorEntry :: CiaoEntry,
    functorPredDefinition :: CiaoPred,
    functorSubfunctorIds :: [String]
  }

newtype CiaoMetaPred = CiaoMetaPred (String, [Int])

newtype CiaoEntry = CiaoEntry (String, [String])

data CiaoPred
  = CPredC CiaoPredC
  | CPredF CiaoPredF
  | EmptyPred

newtype CiaoPredC = CiaoPredC [CiaoClause]

newtype CiaoPredF = CiaoPredF [CiaoFunction]

newtype CiaoBind = CiaoBind (CiaoId, CiaoFunctionBody)

data CiaoFunction = CiaoFunction CiaoHead CiaoFunctionBody [CiaoBind]
```

Figure 2.1: Definition of CiaoSyn (first half)

```
data CiaoFunctionBody
  = CiaoFBTerm CiaoFunctorName [CiaoFunctionBody]
  | CiaoFBCall CiaoFunctionCall
  | CiaoFBLit CiaoLiteral
  | CiaoCaseVar CiaoId [(CiaoFunctionBody, CiaoFunctionBody)]
  | CiaoCaseFunCall CiaoFunctionBody [(CiaoFunctionBody, CiaoFunctionBody)]
  | CiaoEmptyFB

data CiaoFunctionCall = CiaoFunctionCall CiaoFunctorName [CiaoFunctionBody]

data CiaoClause = CiaoClause CiaoHead CiaoBody

type CiaoHead = CiaoTerm

type CiaoBody = [CiaoTerm]

data CiaoTerm
  = CiaoTerm CiaoFunctorName [CiaoArg]
  | CiaoTermLit CiaoLiteral
  | CiaoNumber Int
  | CiaoEmptyTerm

type CiaoFunctorName = CiaoId

newtype CiaoId = CiaoId String

data CiaoArg
  = CiaoArgId CiaoId
  | CiaoArgTerm CiaoTerm

data CiaoLiteral = CiaoLitStr String
```

Figure 2.2: Definition of CiaoSyn (latter half)

The key idea to the whole translation is to build one `CiaoFunctor` for each Haskell function that has to be translated. With that in mind, we would consider a functional program in Ciao (i.e., a `CiaoProgram`) to be a list of `CiaoFunctors`.

For doing that, the topmost module `HsToCiaoPP.hs` takes the Core bindings of each of the Haskell functions, and passes that to the `translate` function (see the MainTranslation.hs file in the repository for the implementation).

After having done that, we now have an internal representation of our translated program. For it to be shown as Ciao source code, it's as simple as endowing each of the CiaoSyn types a `Show` typeclass instance, and writing out the `CiaoProgram` to a file (see the CiaoSyn.hs file in the repository for the implementation).

2.2 Examples of translated programs

Let's take a look at two translation examples:

```
module Tails where

tails :: [Int] -> [[Int]]
tails [] = [[]]
tails (x : xs) = (x : xs) : (tails xs)
```

Figure 2.3: Source code of `Tails.hs`

```
:- module(_,_,[assertions, regtypes, functional, hiord]).

:- meta_predicate tails(?,?).
:- entry tails/2 : {list, ground} * var.
tails([]) := [[]].
tails([X | Xs]) := [[X | Xs] | ~tails(Xs)].
```

Figure 2.4: Source code of `tails.pl`, translated from `Tails.hs`

In this example, the two programs are almost identical, except for the `meta_predicate` and `entry` clauses in the translated source. We'll get to those in section 2.3. This just goes to show that programs in Haskell and functional programs in Ciao are roughly equivalent.

The next example is a bit more interesting.

```
module ListReverse where

import Prelude hiding (reverse)

reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Figure 2.5: Source code of `ListReverse.hs`

```
:- module(_,_,[assertions, regtypes, functional, hiord]).

:- meta_predicate reverse(?,?).
:- entry reverse/2 : {list, ground} * var.
reverse([]) := [].
reverse([X | Xs]) := ~append(~reverse(Xs), [X]).

:- meta_predicate append(?,?,?).
:- entry append/3 : {list, ground} * {list, ground} * var.
append([], X) := X.
append([H | X], Y) := [H | ~append(X,Y)].
```

Figure 2.6: Source code of `listreverse.pl`, translated from `ListReverse.hs`

Most notably, this example shows that while the only user-defined function is `reverse`, the function for appending lists `(++)` also appears in the translation in the form of the `append` predicate. This function was not translated automatically: instead, we have defined it in the `ciao_prelude` (see its definition in the repository here: ciao_prelude.pl), and through the `IDDictionary.hs` module in the translation component, we have specified that the Haskell identifier `(++)` translates to the Ciao identifier `append`. After translating `reverse`, the `Embedder.hs` takes all missing identifiers that the user hasn't defined and fetches them in the `ciao_prelude`.

The `ciao_prelude` and the `IDDictionary.hs` aren't complete, but have been defined to be easily extended as we will mention in section 3.3.

2.3 Cost analysis over translated programs

Performing an analysis over a translated program is done by just calling the corresponding `ciao-shell` script for the specific resource you want to analyze. The script invokes CiaoPP with the corresponding flags for said resource, and outputs its result into a file. After that, our plugin takes that file, parses it through our pretty printing module, and outputs it to another file in a more fitting and user-friendly fashion than CiaoPP's default output, which contains a lot of unnecessary information for the purposes of *hs-to-ciao*. Both the analysis scripts and pretty printer formatters are extensible to support any kind of resource that CiaoPP supports; see section 3.3.

For the resource analysis to work, each Ciao function has to be properly annotated with a `meta_predicate` clause, which indicates the arity of the functions passed as arguments in higher-order functions[1], and an `entry` clause, which provides CiaoPP with information on the arguments so the analysis can be performed properly. Both clauses are generated automatically for user-defined functions in Haskell.

[1]Resource analysis for higher-order functions in CiaoPP aren't supported yet as of this thesis submission.

2.3.1 Examples of Big-O analysis

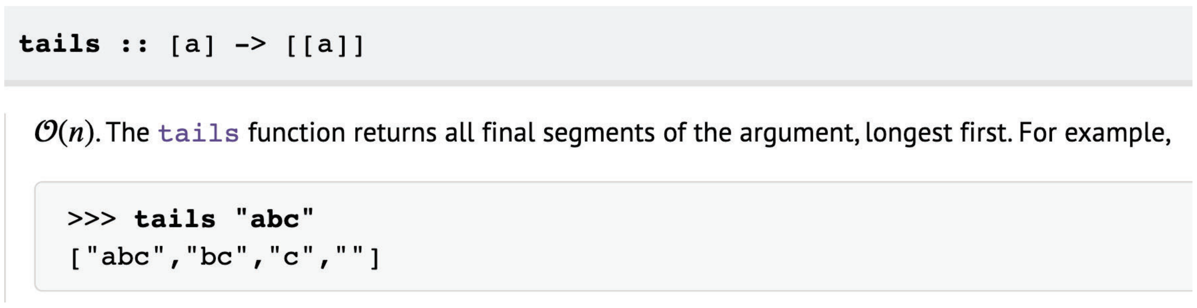
Let's take a look at the output of the Big-O analysis for the examples we showed previously in section 2.2.

```
tails = \a -> length a
```

Figure 2.7: Big-O analysis of `tails.pl`

As you can see, the final output of the Big-O analysis looks quite similar to Haskell's anonymous functions (AKA lambda expressions); the Big-O function for `tails` takes the list `a` as an argument and states that its Big-O function is $O(\texttt{length}(a)) = O(n)$.

This is intuitively true, as the list is only traversed once, but it also matches the documentation in Haskell's `Data.List` module:



```
tails :: [a] -> [[a]]
```

$\mathcal{O}(n)$. The `tails` function returns all final segments of the argument, longest first. For example,

```
>>> tails "abc"
["abc","bc","c",""]
```

Figure 2.8: `tails` documentation for version `base-4.14.0.0`

The Big-O analysis for `listreverse.pl` is:

```
reverse = \a -> (length a)^2

append = \a x -> length a
```

Figure 2.9: Big-O analysis of `listreverse.pl`

This is also correct, given that lists in Haskell are defined as a recursive data type of the form:

```
data List a = [] | Cons a (List a)
```

They are traversed in the same fashion as singly-linked lists. Because of that, you need to traverse the whole list in order to get the last element, which is $O(\texttt{length}(a)) = O(n)$, and you have to do that for each element in the list, i.e. $\texttt{length}(a) = n$ times, which leaves you with the final Big-O function of $(\texttt{length}(a))^2 = O(n^2)$.

Note that the function `append` is also analyzed, since it is used as an auxiliary function in `reverse`. Its analysis is also correct: it receives two lists, `a` and `x` as its arguments, and it's Big-O function only dependes on the `length` of `a`, since you only need to traverse every element of `a` to get to the last one, and after that you just have to update the pointer for the next element to it so that it points to the first element in `x`, hence appending the two lists.

Chapter 3

Results, considerations and conclusions

Having shown the plugin's capabilities and usage, we're now left with discussing a couple of things:

3.1 On the correctness of the plugin

For the translation and analysis to be strictly correct, verifying that the semantics and behavior of the Haskell functions is preserved in the translated Ciao functions would be needed. This was beyond my technical knowhow in formal verification as well as the project's scope and schedule. However, it deserves a mention, as **there aren't any guarantees** that in some corner cases that test some of Haskell's builtin features that Ciao doesn't have, the results of resource analysis hold. However, the results obtained in section 2.2 and subsection 2.3.1 are good enough to show that the approach should be pursued further.

3.2 Existing issues

There are a couple issues existing that we're aware of. For example, the translation will fail if the functions are defined with certain syntactic commodities like point-free style for functions (e.g. `f = (+)` instead of `f x y = x + y`), or certain specific cases of pattern matching. These are unintended, and could most likely be easily fixed.

Also, `entry` clauses for CiaoPP aren't generated properly for higher-order functions; though this is a minor issue given that CiaoPP doesn't support higher-order functions resource analysis yet.

Another limitation that `hs-to-ciao` has is that it can only operate on monomorphic functions; the examples tested involved the types `Int`, `[Int]` and functions using them.

Besides those, there will definitely be more issues that aren't described here as the tool was being actively developed up to the writing of this thesis. If you happen to find a bug, you're more than welcome to open an issue in the GitHub repository for the plugin, and I will try to take a look at it if the project continues its development.

3.3 Extensibility of the plugin

As mentioned several times throughout the thesis, the plugin was designed with extensibility in mind. Other than improving and further developing the plugin's source code, three relatively easy ways to contribute to it are adding analysis scripts and pretty printers, and adding functions to `ciao_prelude`

3.3.1 Analysis scripts

For adding more kinds of analysis, the first thing would be to add a `ciao-shell` script that sets the required flags for CiaoPP in the analysis_scripts folder in the repository. After adding the script, the `KindOfAnalysis` enum type in the AnalysisKinds.hs module has to be extended with a proper name for the kind of analysis that was just added, as well as the functions which rely on that type.

3.3.2 Pretty printing

In case you want to create a different pretty printer for an already existing kind of analysis, or in case you want to create a pretty printer for a new kind of analysis, you just have to add a prettifier as a module in the PrettyPrinters folder, which is a `String -> String` function, which takes as input the bare output that CiaoPP provides and returns the desired output to dump into the corresponding results file.

3.3.3 Adding functions to ciao_prelude

For adding functions into ciao_prelude.pl, you first have to provide the translation for the Haskell identifier for said function in IDDictionary.hs and add the corresponding predicate into `ciao_prelude.pl`, taking into account that you also have to add an `entry` clause for the predicate; documentation on the possible types and properties for them can be found in Ciao's wiki at https://ciao-lang.org/ciao/build/doc/ciao.html/basic_props.html.

3.4 Proposed path forward

In principle, support for functions operating on user-defined algebraic data types is already implemented. However, it hasn't been tested properly, and as such, it's considered as future work, since it's probably not usable in the current state.

The two most important things that remain to do would probably be providing support for higher-order functions and typeclass polymorphism, since they are two key features in real-world Haskell programs. Higher-order functions depend strictly on CiaoPP, as it actually translates and works mostly fine.

Typeclass polymorphism would be the next thing on the roadmap if the project continues, besides fixing existing bugs/issues.

3.5 Personal takeaways

I've loved working on this project, and my stay at IMDEA Software as a whole. As mentioned in the *Acknowledgements*, working under the advisory of Niki has been a really enjoyable and fruitful experience, and Manuel Hermenegildo and the rest of the Ciao team have been nice and ready to lend a hand too when it was needed.

I'm now more than ever convinced that static analysis and functional programming open a whole world of possibilites and useful features when it comes to ensuring that programs are safe and efficient, both of which are two of the most important traits that every piece of software would want and should have.

Chapter 4

Related work

In regards to cost analysis native to Haskell, there have been several different approaches, but most of them require some additional effort on the user side; *Liquid Haskell* [16] requires you to write specific code to reason about your program's resource usage, using refined types for this purpose [5]; however, it also reasons about correctness, whereas our plugin does not (at least no more than what a compilation through GHC would do).

TiML [17] takes an underlying similar approach to CiaoPP, automating the solving of recurrence relations, although it was not designed with Haskell in mind, and you still have to add specific annotations to the code you write.

In comparison with something like AARA [9], which has been extended over the years to support:

- Polynomial bounds [6, 7]
- Parallelism [8]
- Higher-order functions [10]
- Lazy semantics [11]

CiaoPP only provides full support for the first two; it has to be considered that our plugin doesn't support parallelism, though.

In regards to the ways CiaoPP analyzes and other works similar to what we've been doing in the present work, it is using the cost analysis framework developed in [2, 4, 12], based on solving recurrence relations. This framework allows for:

- Backtracking
- Non-determinism
- Failure
- Inference of upper and lower bounds
- Non-polynomial bounds

Among all of these, our plugin takes advantage of the last two, since the first three are concepts that exist within Ciao but not in Haskell (not in the same sense, at least).

Bibliography

[1] A. Serrano, P. Lopez-García, and M. V. Hermenegildo. "Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types". In: *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue* 14.4-5 (2014). Ed. by M. Leushel and T Schrijvers, pp. 739–754. ISSN: 1471-0684. DOI: 10.1017/S147106841400057X. URL: http://cliplab.org/papers/plai-resources-iclp14.

[2] Saumya Debray et al. "Lower Bound Cost Estimation for Logic Programs". In: 1997, pp. 291–305.

[3] E. Mera et al. "Combining Static Analysis and Profiling for Estimating Execution Times". In: *Ninth International Symposium on Practical Aspects of Declarative Languages (PADL'07)*. LNCS 4354. Springer-Verlag, Jan. 2007, pp. 140–154.

[4] R. Haemmerlé et al. "A Transformational Approach to Parametric Accumulated-Cost Static Profiling". In: *Functional and Logic Programming*. Ed. by Oleg Kiselyov and Andy King. Vol. 9613. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 163–180. ISBN: 978-3-319-29603-6 978-3-319-29604-3. DOI: 10.1007/978-3-319-29604-3_11. URL: http://link.springer.com/10.1007/978-3-319-29604-3_11 (visited on 04/19/2020).

[5] Martin A. T. Handley, Niki Vazou, and Graham Hutton. "Liquidate your assets: reasoning about resource usage in liquid Haskell". In: *Proceedings of the ACM on Programming Languages* 4 (POPL Jan. 2020), pp. 1–27. ISSN: 2475-1421, 2475-1421. DOI: 10.1145/3371092. URL: https://dl.acm.org/doi/10.1145/3371092 (visited on 04/18/2020).

[6] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. "Multivariate amortized resource analysis". In: *ACM Transactions on Programming Languages and Systems* 34.3 (Oct. 2012), pp. 1–62. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/2362389.2362393. URL: https://dl.acm.org/doi/10.1145/2362389.2362393 (visited on 04/17/2020).

[7] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. "Resource Aware ML". In: *Computer Aided Verification*. Ed. by P. Madhusudan and Sanjit A. Seshia. Red. by David Hutchison et al. Vol. 7358. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 781–786. ISBN: 978-3-642-31423-0 978-3-642-31424-7. DOI: 10.1007/978-3-642-31424-7_64. URL: http://link.springer.com/10.1007/978-3-642-31424-7_64 (visited on 04/17/2020).

[8] Jan Hoffmann and Zhong Shao. "Automatic Static Cost Analysis for Parallel Programs". In: *Programming Languages and Systems*. Ed. by Jan Vitek. Vol. 9032. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 132–157. ISBN: 978-3-662-46668-1 978-3-662-46669-8. DOI: 10.1007/978-3-662-46669-8_6. URL: http://link.springer.com/10.1007/978-3-662-46669-8_6 (visited on 04/17/2020).

[9] Martin Hofmann and Steffen Jost. "Static prediction of heap space usage for first-order functional programs". In: *ACM SIGPLAN Notices* 38.1 (Jan. 15, 2003), pp. 185–197. ISSN: 03621340. DOI: `10.1145/640128.604148`. URL: `http://portal.acm.org/citation.cfm?doid=640128.604148` (visited on 04/17/2020).

[10] Steffen Jost et al. "Static determination of quantitative resource usage for higher-order programs". In: *ACM SIGPLAN Notices* 45.1 (Jan. 2, 2010), pp. 223–236. ISSN: 0362-1340, 1558-1160. DOI: `10.1145/1707801.1706327`. URL: `https://dl.acm.org/doi/10.1145/1707801.1706327` (visited on 04/17/2020).

[11] Steffen Jost et al. "Type-Based Cost Analysis for Lazy Functional Languages". In: *Journal of Automated Reasoning* 59.1 (June 2017), pp. 87–120. ISSN: 0168-7433, 1573-0670. DOI: `10.1007/s10817-016-9398-9`. URL: `http://link.springer.com/10.1007/s10817-016-9398-9` (visited on 04/17/2020).

[12] P. Lopez-Garcia et al. "Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption". In: *Theory and Practice of Logic Programming* 18.2 (Mar. 2018), pp. 167–223. ISSN: 1471-0684, 1475-3081. DOI: `10.1017/S1471068418000042`. URL: `https://www.cambridge.org/core/product/identifier/S1471068418000042/type/journal_article` (visited on 04/19/2020).

[13] M. Klemen et al. "A General Framework for Static Cost Analysis of Parallel Logic Programs". In: *Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19)*. LNCS. Springer-Verlag, 2020.

[14] M. Klemen et al. "Static Performance Guarantees for Programs with Run-time Checks". In: *20th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'18)*. ACM Press, Sept. 2018.

[15] J. Navas et al. "User-Definable Resource Bounds Analysis for Logic Programs". In: *23rd International Conference on Logic Programming (ICLP'07)*. Vol. 4670. Lecture Notes in Computer Science. Springer, 2007.

[16] Niki Vazou. "Liquid Haskell: Haskell as a Theorem Prover". PhD thesis. UC San Diego, 2016.

[17] Peng Wang, Di Wang, and Adam Chlipala. "TiML: a functional language for practical complexity analysis with invariants". In: *Proceedings of the ACM on Programming Languages* 1 (OOPSLA Oct. 12, 2017), pp. 1–26. ISSN: 2475-1421, 2475-1421. DOI: `10.1145/3133903`. URL: `https://dl.acm.org/doi/10.1145/3133903` (visited on 04/17/2020).