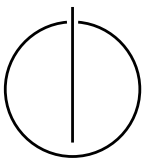
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Automatic Definition of Running Time Functions

Jonas Stahl



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

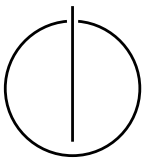
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Automatic Definition of Running Time Functions

Automatische Definition von Laufzeitfunktionen

Author:	Jonas Stahl
Supervisor:	Prof. Dr. Tobias Nipkow
Advisor:	Prof. Dr. Tobias Nipkow
Submission Date:	16.02.2024



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 16.02.2024 Jonas Stahl

Acknowledgments

Abstract

Contents

1 Introduction

1.1 Section

Citation test [Lam94].

Acronyms must be added in `main.tex` and are referenced using macros. The first occurrence is automatically replaced with the long version of the acronym, while all subsequent usages use the abbreviation.

E.g. `\ac{TUM}, \ac{TUM}` ⇒ Technical University of Munich (TUM), TUM

For more details, see the documentation of the `acronym` package[1].

1.1.1 Subsection

See Table 1.1, Figure 1.1, Figure 1.2, Figure 1.3.

Table 1.1: An example for a simple table.

A	B	C	D
1	2	1	2
2	3	2	3

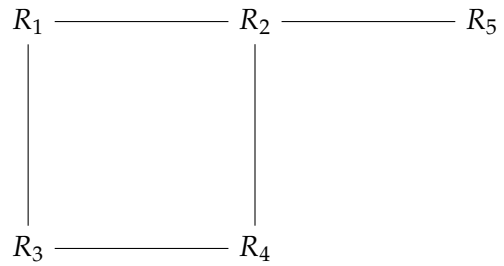


Figure 1.1: An example for a simple drawing.

[1] `https://ctan.org/pkg/acronym`

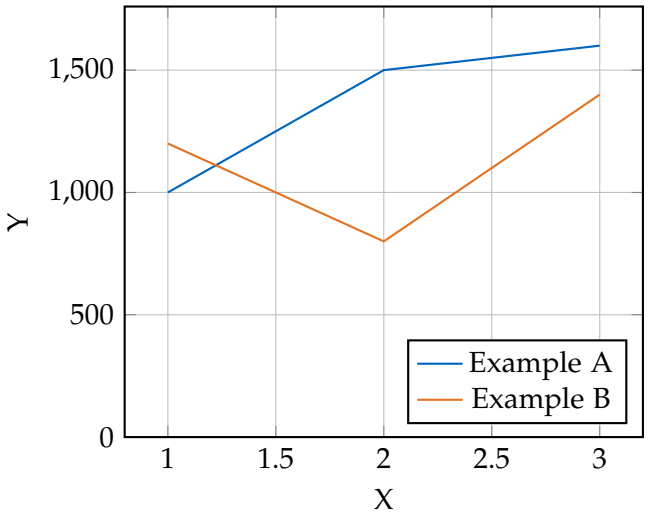


Figure 1.2: An example for a simple plot.

```
SELECT * FROM tbl WHERE tbl.str = "str"
```

Figure 1.3: An example for a source code listing.

2 Related work

This chapter will look at existing approaches to define running time functions. Therefore we start with a reduced version for First Order Functions and extends it to support Higher Order as well as Curried function. All schemas are built to given an approximation for the number of function calls used.

2.1 First Order Functions

At first we gonna look at First Order Function. Here, no function are allowed as arguments. Also the use of curried functions is forbidden, this means every function has to be fully evaluated in one application.

The first scheme presented first was used by Sands in "Calucli for Time Analysis of Functional Programs" [San90]. His key idea was splitting up the functions into primitive (p) and non-primitive functions (f). Als primitive function should be free of cost, this should include basic mathematical operations and comparisons. When calling non-primitive function the timing function should be evaluated instead. In both cases also the arguments need to be transformed and added. Constants (c) are also seen as free and therefore translated as 0. In If-Else constructs it would be possible to just add up all the used terms, but this would lead to a big overapproximation. To get a better result, the relevant branch along with the condition is counted. The function translating this the expressions is named $\mathcal{T}$.

When converting a function definition, we start with 1 for the function call itself. Afterwards the expression converted by $\mathcal{T}$ is added. The conversion function is named $\mathcal{C}$. The full schema for translating expressions is shown in listing 2.1. Sands also provides as proof for this schema. A formalization can be found in chapter 3.

$$\mathcal{T}[\![f\ a_1\ \ldots\ a_n]\!] = (T\_f\ a_1\ \ldots\ a_n + \mathcal{T}[\![a_1]\!] + \ldots + \mathcal{T}[\![a_n]\!])$$
$$\mathcal{T}[\![p\ a_1\ \ldots\ a_n]\!] = (\mathcal{T}[\![a_1]\!] + \ldots + \mathcal{T}[\![a_n]\!])$$
$$\mathcal{T}[\![c]\!] = 0$$
$$\mathcal{T}[\![\texttt{IF}\ c\ \texttt{THEN}\ e_1\ \texttt{ELSE}\ e_2]\!] = (\mathcal{T}[\![c]\!] + \texttt{IF}\ c\ \texttt{THEN}\ \mathcal{T}[\![e_1]\!]\ \texttt{ELSE}\ \mathcal{T}[\![e_2]\!])$$

$$\mathcal{C}[\![f\ a_1\ \ldots\ a_n = e]\!] = (f\ a_1\ \ldots\ a_n = 1 + \mathcal{T}[\![e]\!])$$

Listing 2.1: Translation schema for first order functions by Sands

The schema introduced by Nipkow in "Functional Data Structures" slightly different approach regarding primitive functions and constants [Nip23]. At first every primitive function, here defined as basic mathematical operations and constructors, as well as constants are translated as 1. In a simplification step all $+1$ except one are dropped, as they don't change the asymptotic running time. This results in the same schema as given by Sands. Additionally it gives a schema to translate case and let expressions. Case works similar to if-else, while the body of let expressions are being converted to lambdas. (#TODO this might change) The schema can be found in listing 2.2.

$$\mathcal{T}[\![\texttt{case } e \texttt{ of } p_1 \Rightarrow e_1 \texttt{ | } \ldots \texttt{ | } p_k \Rightarrow e_k]\!]$$
$$= \mathcal{T}[\![e]\!] \texttt{ + (case } e \texttt{ of } p_1 \Rightarrow \mathcal{T}[\![e_1]\!] \texttt{ | } \ldots \texttt{ | } p_k \Rightarrow \mathcal{T}[\![e_k]\!]\texttt{)}$$
$$\mathcal{T}[\![\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2]\!] = \mathcal{T}[\![e_1]\!] \texttt{ + } (\lambda x.\mathcal{T}[\![e_2]\!])\ e_1$$

Listing 2.2: Translation schema for case- and let-expression by Nipkow

2.2 Higher Order functions

Extension by Sands

2.3 Curried Functions

Extension by Sands

3 Formalization

The following chapter explaines the formalization of the proof provides by Sands about the translation schema for a first-order language [San90]. The whole theory file can be found inside the GitHub repository [#TODO link].

He start by defining a simple imperative language we gonna translate. The expressions should work under arbitrary types while still providing a numeral type we can use for counting the steps. Therefore we define the type value which is not further specified. We now define the datatype using natural number as well as the not specified datatype. As a result we obtain a datatype satisfying our conditions.

```
typedecl "value"
datatype val = N nat | V "value"
```

We later also need a boolean interpretation of our datatype. We define false for the Natural type with 0 and everything else as true.

```
definition false :: "val ⇒ bool" where "false v ≡ v = N 0"
definition true :: "val ⇒ bool" where "true v ≡ ¬ false v"
```

The proof differences between primitive functions without costs and non-primitive function costing 1 per call. In the followig non-primitive function will only be called functions. We define datatype representing an identifier for both of them. Additionally, functions need to differ between a function and a timing function to avoid collisions in the namspace.

```
datatype funId =
  Fun string
| cFun string
datatype pfunId = pFun string
```

With everything defined we can now provide a definitions for an expression. For better readability we introduce $ for function application and $$ for primitive function applications. The whole datatype can be found in listing 3.1.

All functions will be stored in a definition mapping a function id to an expression option. If the mapper returns None then no function is defined. This is needed later again for avoid namespace collisions.

```
datatype exp =
  App funId "exp list" (infix "\$" 100)
| pApp pfunId "exp list" (infix "\$\$" 100)
| If exp exp exp ("(IF _/ THEN _/ ELSE _)")
| Ident nat
| Const val
```

Listing 3.1: Expression syntax

```
type_synonym defs = "funId ⇒ exp option"
```

For primitive function there's no mapper but a direct primitive application function. It take the name of the function and a list of arguments. The function should be universal to support an kinds of primitive functions on different machines. Sands describes it to contain

There is also a definition for primitive functions. Similar for the datatype we want to keep them as universal as possible. The function is only assumes to know the function sum with arbitrary many arguments as we need it to define the timing functions later. The function can be assumed to be primitive as the number of arguments is fixed in every expression.

```
axiomatization pApp :: "string ⇒ val list ⇒ val" where
  sum: "pApp ''sum'' es = (N o sum_list o map val_to_nat) es"
```

The last definition need for simulating a function application is an environment containing the local variables. Those will be represented by a list of values. The values can be referred through the expression Ident by giving the index of the variable.

```
type_synonym env = "val list"
```

We now defined the evaluating semantic of the following form.

$$\rho, \phi \vdash \mathrm{e} \to \mathrm{v}$$

It can be read the following: *The expression e evaluates under the local variables ρ and the definitions ϕ to the value v*. The semantic of eval is given in figure 3.2.

For the semantic we can show determinism given through the following formula. The proove is an induction over eval which can be solved mostly automatically by metis and blast.

$$[\![\rho, \phi \vdash e \to v;\ \rho, \phi \vdash e \to v']\!] \Longrightarrow v = v'$$

```
inductive eval :: "env ⇒ defs ⇒ exp ⇒ val ⇒ bool"
  ("(_/, _/ ⊢ _/ → _)") where
Id: "ρ, ϕ ⊢ Ident i → (ρ ! i)" |
C: "ρ, ϕ ⊢ Const v → v" |
F: "length es = length vs
  ⟹ (∀i < length vs. ρ, ϕ ⊢ (es ! i) → (vs ! i))
  ⟹ ϕ f = Some fe ⟹ vs, ϕ ⊢ fe → v
  ⟹ ρ, ϕ ⊢ (f$ es) → v" |
P: "length es = length vs
  ⟹ (∀i < length es. ρ, ϕ ⊢ (es ! i) → (vs ! i))
  ⟹ pApp p vs = v ⟹ ρ, ϕ ⊢ (pFun p$$ es) → v" |
If1: "ρ, ϕ ⊢ b → v ⟹ true v
  ⟹ ρ, ϕ ⊢ t → et ⟹ ρ, ϕ ⊢ (IF b THEN t ELSE f) → et" |
If2: "ρ, ϕ ⊢ b → v ⟹ false v
  ⟹ ρ, ϕ ⊢ f → ef ⟹ ρ, ϕ ⊢ (IF b THEN t ELSE f) → ef"
```

Listing 3.2: Default evaluation semantic

In the next step we define a step-counting semantic of the form

$$\rho, \phi \vdash e \rightarrow_s (v, t)$$

which is read as: *The expression e evaluates under the given local variables ρ and the definitions ϕ to the value v.* The corresponding semantic is given in figure 3.3.

The now want to show the equivalence of the semantic according to the resulting value. Showing the implication from the step-counting semantic to the normal semantic can be done by induction over the step-counting semantic. All cases can be handled by auto.

```
lemma eval_count_eval: "ρ, ϕ ⊢ b →s (v,t) ⟹ ρ, ϕ ⊢ b → v"
by (induction ρ ϕ b "(v,t)" arbitrary: v t rule: eval_count.induct) auto
```

The other direction also is not difficult but needs an lengthy axiliary lemma to help Isabelle dealing with the arguments of the function applications. The lemma states the following:

```
lemma eval_eval_count':
  assumes tex:"∀i < length vs. ∃t. ρ, ϕ ⊢ es ! i →s (vs ! i, t)"
    and   len: "length es = length vs"
   shows  "∃ts. length vs = length ts
           ∧ (∀i. (i < length vs ⟶ ρ, ϕ ⊢ (es!i) →s (vs!i,ts!i)))"
```

```
inductive eval_count :: "env ⇒ defs ⇒ exp ⇒ val * nat ⇒ bool"
  ("(_/, _/ ⊢ _/ →s _)") where
cId: "ρ, ϕ ⊢ Ident i →s (ρ!i,0)" |
cC: "ρ, ϕ ⊢ Const v →s (v,0)" |
cF: "length es = length vs ⟹ length es = length ts
  ⟹ (∀ i < length vs. ρ, ϕ ⊢ (es!i) →s (vs!i,ts!i))
  ⟹ ϕ f = Some fe ⟹ vs, ϕ ⊢ fe →s (v,t)
  ⟹ ρ, ϕ ⊢ (f$ es) →s (v,1+t+sum_list ts)" |
cP: "length es = length vs ⟹ length es = length ts
  ⟹ (∀i < length es. ρ, ϕ ⊢ (es ! i) →s (vs!i,ts!i))
  ⟹ pApp p vs = v ⟹ ρ, ϕ ⊢ (pFun p$$ es) →s (v,sum_list ts)" |
cIf1: "ρ, ϕ ⊢ b →s (eb,tb) ⟹ true eb ⟹ ρ, ϕ ⊢ t →s (et,tt)
  ⟹ ρ, ϕ ⊢ (IF b THEN t ELSE f) →s (et,tb+tt)" |
cIf2: "ρ, ϕ ⊢ b →s (eb,tb) ⟹ false eb ⟹ ρ, ϕ ⊢ f →s (ef,tf)
  ⟹ ρ, ϕ ⊢ (IF b THEN t ELSE f) →s (ef,tb+tf)"
```

Listing 3.3: Step-Counting Semantic

The lemma is shown as an induction over the length of vs with arbitrary es using the assumptions. While the Nil case can be handled be simp, the Cons case needs some more effort. We receive the assumptions as premises for (v#vs) as well as the induction hypothesis (IH). To use the IH we need to use the premise of tex and that we can shift it down and also holds for v. Same can be done for len. As a result we can now use the IH and obtain the ts wanted for vs. From tex we can obtain the t corresponding to v. In the last step we only need to combine the two results and show the expected result. With the help of it is now possible to show the direction from eval semantic to step-counting semantic by induction of the eval semantic. This gives us the equivalance of the two semantics according to the value result.

Sands also claims in a proposition, that the counting result of the step-counting semantic equals the count the rule F was used in the eval semantic. As this is quite laborious to do in Isabelle while only providing a proof for an trivial step the proof is neglected.

We can now define the main function converting an expression to its cost equivalent. The used schema corresponds to the one presented in #TODO.

```
fun 𝒯 :: "exp ⇒ exp" where
  "𝒯 (Const v) = Const (N 0)"
| "𝒯 (Ident i) = Const (N 0)"
| "𝒯 (IF b THEN t ELSE f)
```

```
   = pFun ''sum''$$ [𝒯 b, IF b THEN 𝒯 t ELSE 𝒯 f]"
| "𝒯 (pFun _$$ args) = pFun ''sum''$$ map 𝒯 args"
| "𝒯 (Fun f$ args) = pFun ''sum''$$ (cFun f$ args # map 𝒯 args)"
```

now We add a definition for converting a function expression. It uses the $\mathcal{T}$ function and adds $1+$ to the front. Based on this function, we also define the function conv. It accepts a definition and returns a definition, but it will always return the timing function of a corresponding defined function. Already defined timing functions will be dropped. #TODO reformulate this paragraph

```
definition cost :: "exp ⇒ exp" where
  "cost e = pFun ''sum'' [Const (N 1), 𝒯 e]"

fun conv :: "defs ⇒ defs" where
  "conv ϕ (Fun f) = ϕ (Fun f)"
| "conv ϕ (cFun f) = (case ϕ (Fun f) of None ⇒ None
                                      | Some e ⇒ Some (cost e))"
```

For the final correctness definition we still need a property claiming that in a given definition no timing function is defined. This avoids errors of timing functions being dropped by the conv function.

```
definition no_time where
  "no_time ϕ = (∀f. ϕ (cFun f) = None)"
```

We also define a helping lemma stating that if there is no timing function in the definition then the evaluated result of an expression does not change when converting a definition using conv. It is proven over induction with a helping lemma claming conv does not change defined functions.

```
lemma no_time_trans:
  "no_time ϕ ⟹ ϕ f = Some e ⟹ (conv ϕ) f = Some e"
  by (cases f) (auto simp: no_time_def)
lemma eval_no_time_trans:
  "ρ, ϕ ⊢ e → v ⟹ no_time ϕ ⟹ ρ, (conv ϕ) ⊢ e → v"
  by (induction rule: eval.induct) (auto simp: no_time_trans)
```

We can now show the final correctness lemma. Given an expression and a definition without timing function. If there exists an evaluation of the step-counting semantics to count t then evaluation the converted expression and timing function results to the same value t.

```
theorem conv_cor:
  assumes "ρ, ϕ ⊢ e →s (s,t)"
```

```
  and "no_time ϕ"
shows "ρ, (conv ϕ) ⊢ (𝒯 e) → N t"
```

The theorem can be prooved by induction over the step-counting semantic. Only the simplest cases Id and C can be solved automatically. The others need an Isar proof. In the following the take a look at the most important case F used for applying a function. We have the following assumptions.

(1) length es = length vs = length ts

(2) ∀i<length vs. $\rho, \phi \vdash$ es ! i $\rightarrow_s$ (vs ! i, ts ! i) ∧ (no_time $\phi \longrightarrow \rho$, conv $\phi \vdash \mathcal{T}$ (es ! i) $\rightarrow$ N (ts ! i))

(3) ϕ f = Some fe

(4) vs, $\phi \vdash$ fe $\rightarrow_s$ (v,t)

(5) no_time ϕ

(6) vs, conv $\phi \vdash \mathcal{T}$ fe $\rightarrow$ N t

(6) is already collapsed as the premise is given by (5). The cost for the function application consists of the cost for evaluation the time function and evaluating the arguments for the function. We start with the cost for evaluating the time function. By using the rule for primitive application of the evaluation semantic we can conclude the following equation

$$\text{vs, conv } \phi \vdash \text{(pFun sum)\$\$ [Const (N 1), } \mathcal{T}\text{fe]} \rightarrow \text{ N (1 + t).}$$

The needed premises can be derived from (6) and the axiom for primitive functions. From here we only need to use the definition of the cost function to have a term about the cost term of the function.

$$\text{vs, conv } \phi \vdash \text{cost fe} \rightarrow \text{ N (1 + t).} \tag{3.1}$$

In the next step we want to show, that the result stays the same if the cost function of our function is called. For this we need the evaluation of the arguments es for the function. The evaluated result vs is already given for the step-counting evaluation. Here we can use the lemma eval_count_eval stating that the result is the same for the normal semantic.

$$\forall \text{ i<length vs. } \rho, \phi \vdash \text{es ! i} \rightarrow \text{vs ! i}$$

Additionally we use the lemma eval_no_time_trans with (5) an can conclude that the result also stays the when the function definitions are converted.

$$\forall \text{ i<length vs. } \rho, \text{conv}\phi \vdash \text{es ! i} \rightarrow \text{vs ! i} \tag{3.2}$$

From (3) and (5) we can conclude that f is an identifier for timing functions, therefore

$$\text{conv } \phi \text{ f} = \text{Some fe}$$

holds. Using this with (1) and the equation 3.2 we can use the rule for function application to gain

$$\rho, \text{conv } \phi \vdash \text{ cFun fn\$ es } \rightarrow \text{ N (1 + t).} \tag{3.3}$$

The next step involves combining 3.3 into one list with the cost evaluation of the arguments. From (2) with (1) and (5) the mapping of the arguments can be deduced.

$$\forall \text{i<length (map } \mathcal{T} \text{ es). } \rho, \text{conv } \phi \vdash \text{ (map } \mathcal{T} \text{ es) ! i } \rightarrow \text{ (map N ts) ! i}$$

In combination with equation 3.3 we get the full list of arguments for the sum function.

$$\begin{aligned}&\forall \text{i<length (cFun fn\$ es \# map } \mathcal{T} \text{ es).}\\ &\rho, \text{conv } \phi \vdash \text{ (cFun fn\$ es \# map } \mathcal{T} \text{ es) ! i } \rightarrow \text{ ((N (1 + t)) \# (map N ts)) ! i}\end{aligned}$$

With this, (1) and the axiom for sum we can now use the rule for primitive function application and show the wanted result.

$$\rho, \text{conv } \phi \vdash \mathcal{T} \text{ (f \$ es) } \rightarrow \text{ N (1 + t + sum\_list ts)}$$

This completes the proof for this case. The other cases can be found in the appended theoy file. #TODO wo auch immer des hin kommt

4 Implementation

Implementenation fun

5 Summary

Summarying everything

Abbreviations

TUM Technical University of Munich

List of Figures

List of Tables

Bibliography

[Lam94] L. Lamport. *LaTeX : A Documentation Preparation System User's Guide and Reference Manual*. Addison-Wesley Professional, 1994.

[Nip23] T. Nipkow. "Functional Data Structures and Algorithms A Proof Assistant Approach." In: (2023).

[San90] D. Sands. "Calculi for time analysis of functional programs." PhD thesis. University of London, 1990.