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

Abstract

Contents

Abstract **iv**

1 Introduction **1**

2 Related work **2**
2.1 First Order Functions 2
2.2 Higher Order functions 3
2.3 Curried Functions 6

3 Formalization **9**

4 Implementation **16**
4.1 Commands 16
4.2 Schema 17
4.3 Termination proof 20
4.4 Restrictions 22
4.4.1 Functions in datatypes 22
4.4.2 Operations on datatypes 22
4.4.3 Partial application 22
4.5 Probably examples 23

5 Summary **24**

List of Figures **25**

Bibliography **26**

1 Introduction

To examine new datastructures and algorithms running time is one of the most important measures. In order to determine the running time of a function we first need to convert them. Afterwards the we can use proof assistants as Isabelle to proof a certain running time. Currently this first conversion step is done manually in Isabelle. As all human tasks this can create errors and make the following proof worthless. Goal of this work is to create a automatic converter from functions into their running time function for Isabelle. The converted function will represent the number of function calls inside of a function evaluation. To simplify the resulting function and therefore also proofs we only consider the running time class as relevant. This means we will apply some improvements and drop constants.

Starting with chapter 2 we take a look at the default conversion schema used in other literature. We start with a schema for more restricted functions and relax the restrictions till with see a schema for curried higher-order functions. To proof the correctness of the used schema we gonna look at a proof for the conversion schema in chapter 3. The proof was formalized in Isabelle. In chapter 4 we take a look at the actual implementation. The chapter gives an overview on how to use the command and explains the used termination proof. In the end it also gives an overview on the current restrictions the converter cannot be used for.

#TODO isabelle version number

2 Related work

This chapter will look at the schemas used to define running time functions. We start by first looking at non-curried first-order functions. Afterward, we extend this schema for higher-order functions and continue by allowing curried functions. All schemas count the number of function calls made. Also, we are only interested in the time complexity and, therefore, ignore constants.

2.1 First Order Functions

We restrict our language to first-order functions without curried function applications for the first schema. In other words, no function is allowed as an argument, and every function application has to evaluate the function entirely.

All found papers use quite similar schemas. Therefore, we only look at two sources as they cover all the cases we want to cover in this work. The scheme presented first was used by Sands in "Calucli for Time Analysis of Functional Programs" [San90]. The schema is built up in the following way. He starts by splitting up the functions into primitive (p) and non-primitive functions (f). All primitive functions should only take a constant amount of time and, therefore, be cost-free for our evaluation. They include basic mathematical operations and comparisons. The cost for calling a non-primitive function is represented by its timing function. As a result, we need to transform it by replacing it with its timing function instead. In both cases, we also need to add the cost of the arguments. Constants (c) are translated as 0 because we are only interested in the time complexity. In If-Else constructs, it would be possible to add up all the used terms, but this would lead to a big overapproximation. To get a better result, we only add up the cost for the condition, which is always evaluated, with the cost for the relevant branch. The translation function is named $\mathcal{T}$ and is described in figure 2.1.

When converting a function definition, we start with 1 for the function call itself. Afterward, the expression converted by $\mathcal{T}$ is added. The conversion function is named $\mathcal{C}$ and is described in figure 2.2. Sands also provides a proof for this schema. Chapter 3 presents a formalization of the proof in Isabelle.

The schema introduced by Nipkow in "Functional Data Structures" provides the same translation for the cases presented so far [Nip23]. Additionally, he defines a schema to translate case and let expressions. We look at them here as we will need them for the

$$
\begin{aligned}
\mathcal{T}[\![f\ a_1\ \ldots\ a_n]\!] &= (T\_f\ a_1\ \ldots\ a_n + \mathcal{T}[\![a_1]\!] + \ \ldots\ + \mathcal{T}[\![a_n]\!]) \\
\mathcal{T}[\![p\ a_1\ \ldots\ a_n]\!] &= (\mathcal{T}[\![a_1]\!] + \ \ldots\ + \mathcal{T}[\![a_n]\!]) \\
\mathcal{T}[\![c]\!] &= 0 \\
\mathcal{T}[\![\texttt{IF}\ c\ \texttt{THEN}\ e_1\ \texttt{ELSE}\ e_2]\!] &= (f\ a_1\ \ldots\ a_n = 1 + \mathcal{T}[\![e]\!])
\end{aligned}
$$

Figure 2.1: Translation function $\mathcal{T}$ for expressions of first-order functions

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

Figure 2.2: Translation function $\mathcal{C}$ for function definitions of first-order functions

implementation. Similar to if-else constructs, we avoid too much overapproximation by only counting the executed case. To this, we add the running time equivalent of the pattern-matching expression. For the let expression, we add the running time version of the assigned expression to the let expression with the translated body. If the translated body does not use the assigned variable, we can reduce the construct to the body. The schema can be found in listing 2.1.

$$
\begin{aligned}
&\mathcal{T}[\![\texttt{case}\ e\ \texttt{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k]\!] \\
&\quad = \mathcal{T}[\![e]\!] + (\texttt{case}\ e\ \texttt{of}\ p_1 \Rightarrow \mathcal{T}[\![e_1]\!] \mid \ldots \mid p_k \Rightarrow \mathcal{T}[\![e_k]\!]) \\
&\mathcal{T}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] = \mathcal{T}[\![e_1]\!] + (\texttt{let}\ x = e_1\ \texttt{in}\ \mathcal{T}[\![e_2]\!])
\end{aligned}
$$

Listing 2.1: Translation schema for case- and let-expression by Nipkow

2.2 Higher Order functions

In this chapter, we want to extend the schema for higher-order functions. First, we look at the function `test` in listing 2.2.

```
fun test :: "(nat ⇒ bool) ⇒ nat ⇒ nat" where
  "test f a = (if f a then 1 else 0)"
```

Listing 2.2: Example function

Applying the translation $\mathcal{T}$ for first-order functions onto the expression yields us

```
T_f a + (if f a then 0 else 0).
```

$$f' \; x_1 \; \ldots \; x_n = \mathcal{V}[\![e]\!]$$
$$cf \; x_1 \; \ldots \; x_n = 1 + \mathcal{N} o \mathcal{V}[\![e]\!]$$

Figure 2.3: Translation schema for higher-order function with expression e

This example shows that we need a way to address the original function as well as the timing function. As we cannot generate the timing function at runtime, Sands proposes to change every function argument by a pair [San90]. The first element is the original function and the second is the timing function. Using this idea, we can convert the whole function as shown in listing 2.3.

```
fun T_test :: "('a ⇒ bool) × ('a ⇒ nat) ⇒ 'a ⇒ nat" where
  "T_test f a = 1 + snd f a + (if fst f a then 1 else 0)"
```

Listing 2.3: Timing function of example function

In general, we need to use `snd` everywhere the timing function is needed and `fst` in case of a normal evaluation. For every function `f` he generates two functions `f'` and `cf`. The function `f'` evaluates to the result of the original function but with the described pair instead of functions as an argument. The function `cf` is the timing function, called cost function by Sands. The translation schema is described in figure 2.3.

He uses the two referenced functions $\mathcal{V}$ and $\mathcal{T}$ for conversion. The function $\mathcal{V}$ replaces every function identifier by the described pair. Therefore, it ensures that every function will be called with the expected pair. The function `fst` is used for every function application. The result behaves as described for the function f'. The schema is described in figure 2.4. For the timing function, the function $\mathcal{T}$ is applied afterward. It behaves similarly to the schema described for the first-order function in figure 2.1. The only difference is that the `snd` function is used in every application to get the timing function. The schema is written in figure 2.5. Function definitions will not be converted to a pair and back by those functions. This optimization step could be dropped but keeps the timing function readable.

Nipkow provides no schema for higher-order functions. However, as he also needs some higher-order functions for the analysis, he provides a translation. In listing 2.4, the translation for the `map` function is shown. It differs from our schema as it only takes the timing function but not the original function. This schema works as the original function is not used but cannot be extended to all functions.

```
fun map :: "('a ⇒ 'b) ⇒ 'a list ⇒ 'b list" where
  "map f [] = []"
| "map f (x # xs) = f x # map f xs"
```

$$
\begin{aligned}
\mathcal{V}[\![f\ a_1\ \dots\ a_n]\!] &= f'\ \mathcal{V}[\![a_1]\!]\ \dots\ \mathcal{V}[\![a_n]\!] \\
\mathcal{V}[\![p\ a_1\ \dots\ a_n]\!] &= p\ \mathcal{V}[\![a_1]\!]\ \dots\ \mathcal{V}[\![a_n]\!] \\
\mathcal{V}[\![c]\!] &= c \\
\mathcal{V}[\![\text{IF } c \text{ THEN } e_1 \text{ ELSE } e_2]\!] &= \text{IF } \mathcal{V}[\![e_1]\!] \text{ THEN } \mathcal{V}[\![e_2]\!] \text{ ELSE } \mathcal{V}[\![e_2]\!] \\
\mathcal{V}[\![f]\!] &= (f', cf) \\
\mathcal{V}[\![p]\!] &= (p, cp) \\
\mathcal{V}[\![e\ a_1\ \dots\ a_n]\!] &= \texttt{fst}\ \mathcal{V}[\![e]\!]\ \mathcal{V}[\![a_1]\!]\ \dots\ \mathcal{V}[\![a_n]\!]
\end{aligned}
$$

Figure 2.4: Translation schema V for higher-order functions

$$
\begin{aligned}
\mathcal{T}[\![f'\ a_1\ \dots\ a_n]\!] &= \mathcal{T}[\![a_1]\!] + \cdots + \mathcal{T}[\![a_n]\!] + cf\ a_1\ \dots\ a_n \\
\mathcal{T}[\![p\ a_1\ \dots\ a_n]\!] &= \mathcal{T}[\![a_1]\!] + \cdots + \mathcal{T}[\![a_n]\!] \\
\mathcal{T}[\![c]\!] &= 0 \\
\mathcal{T}[\![\text{IF } c \text{ THEN } e_1 \text{ ELSE } e_2]\!] &= \mathcal{T}[\![c]\!] + \text{IF } c \text{ THEN } \mathcal{V}[\![e_1]\!] \text{ ELSE } \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![\texttt{fun}\ e\ a_1\ \dots\ a_n]\!] &= \mathcal{T}[\![e]\!] + \mathcal{T}[\![e_1]\!] + \cdots + \mathcal{T}[\![e_n]\!] + \texttt{cost}\ e\ e_1\ \dots\ e_n \\
\mathcal{T}[\![(f', cf)]\!] &= 0 \\
\mathcal{T}[\![(p, cp)]\!] &= 0
\end{aligned}
$$

Figure 2.5: Translation T for higher-order functions

```
fun T_map :: "('a ⇒ nat) ⇒ 'a list ⇒ nat" where
  "T_map f [] = 1"
| "T_map f (x # xs) = 1 + (f x + T_map f xs)"
```

Listing 2.4: Translation of the function map to their timing function by Nipkow

2.3 Curried Functions

Sands also specifies a schema for curried functions. It operates assuming that the application is free of cost as long as the function is not fully applied. The key point of this schema is the extension of the pair used for higher-order function arguments to a triple. The third argument represents the number of arguments left to apply till full application. This third argument is called the arity. He explains the schema by demonstrating it on a default apply function first. The apply function `app` is shown in listing 2.5.

```
fun app :: "('a ⇒ 'b) ⇒ 'a ⇒ 'b" where
  "app f a = f a"
```

Listing 2.5: Apply function

He also starts by defining the function `app'`. This function accepts the described triple instead of the function. If the arity equals 1, then the curried function expects only one more argument. Then, the first element of the tuple representing the original function is evaluated. If the arity is bigger than 1, the curried function will evaluate to another function. Therefore, we construct a new tuple with the argument applied to both functions and the arity decreased by 1. The full function is described in listing 2.6.

```
fun app' where
  "app' f x = (if arity f = 1
      then (func f x)
      else (func f x, cost f x, arity f - 1))"
```

Listing 2.6: Apply function on function argument triple

Afterward, he defines the function `capp`. It should yield the cost of the passed function with one argument applied. As the application is free if the function is not fully applied, it yields 0 if the arity is greater than 1. If the arity is 1, then the cost function is evaluated. This gives the function definition shown in listing 2.7.

```
fun capp where
```

$$\mathcal{T}[\![f \mathbin{@} a]\!] = \mathcal{T}[\![f]\!] + \mathcal{T}[\![a]\!] + (f \ c@ \ a)$$

Figure 2.6: Converting application in case of curried higher-order functions

```
"capp f x = (if arity f = 1 then cost f x else 0)"
```

Listing 2.7: Timing function of apply function

Similar to the translation of the higher-order function, he defines two conversion functions $\mathcal{V}$ and $\mathcal{T}$ with similar behavior. $\mathcal{V}$ converts the function to evaluate as the original function but accepts the described triples instead of functions. Every referenced function will be replaced by the described triple. Function applications are replaced by passing function and argument to the `app'` function. All the other definitions stay the same as described in 2.4. The function $\mathcal{T}$ also stays the same as in 2.5 except for the application case. For every application currently handled by the `app'` function, the function and the argument will be converted and added up. Additionally, we need the cost of the application itself. This cost is determined by passing the function and the argument to the function `capp`. Figure 2.6 shows this case. For better readability, the infix symbol @ represents the function `app'` while $c@$ stands for the function `capp`. For example, we can see the transformation of the map function in listing 2.8.

```
fun map where
  "map f [] = []"
| "map f (x#xs) = f x # map f xs"

fun map' where
  "map' f [] = []"
| "map' f (x#xs) = (Cons,T_Cons,2) @ (f @ x) @ ((map',T_map,2) @ f @ xs)"

fun T_map where
  "T_map f [] = 1"
| "T_map f (x#xs) = 1 + f c@ x +
    (map',T_map,2) c@ f + (map',T_map,2) @ f c@ xs +
    (Cons,T_Cons,2) @ (f @ x) c@ ((map',T_map,2) @ f @ xs)"
```

Listing 2.8: Example translation for map function

As we can see, this already blows up for a small function as `map`. To overcome this issue, Sands introduces some basic optimization steps. If the `capp` function is applied onto a triple, an application to a defined function happens. We differ between two cases here. If the arity is greater than 1, it can be dropped as it would evaluate to 1 either. In case the arity is 1, we can replace this by the application to the cost function. For the normal application `app'`, he applies the same but evaluates the normal function instead of the cost function. As a result, we gain a timing function similar to our previous schema. The apply function will only be used for passed functions now.

```
function T_map where
  "T_map f [] = 1"
| "T_map f (x#xs) = 1 + f c@ x + T_map f xs"
```

3 Formalization

Sands provides a correctness proof for all translation schemas presented in chapter 2. This chapter looks at the correctness proof for a first-order language. The proof was formalized in Isabelle and can be found in the associated GitHub repository [#TODO link]. The term cost function is used interchangeably for timing function here.

We start by defining the first-order imperative language. The datatype `val` represents the type our language operates over. It needs to fulfill two requirements. The language should work with arbitrary types we do not want to define yet. In order to count the number of function calls, we need a type that can count them. Therefore, we define a type `value`, which is not further specified. This represents our arbitrary type. Based on this, we define the constructors for our datatype. The `V` constructor maps the arbitrary type and the `N` constructor wraps the natural numbers we will use for counting. Therefore it fulfills our requirements.

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

For if-else constructs, we need a boolean interpretation of our datatype. We define the natural number type with 0 as false and everything else as true.

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

As explained in the last chapter, we differ between primitive and non-primitive functions. The call of a primitive function is free, while a non-primitive function call costs 1. In the following, we will refer to non-primitive functions only as functions. For each of them, we now define a datatype representing an identifier. For functions, we also need to differentiate between an identifier for a function and its respective cost function. This distinction avoids collisions in the namespace for the final correctness proof.

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

```
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

We can now define expressions. An application takes a list of arguments, each represented by another expression. For better readability, we introduce $ for function applications and $$ for primitive function applications. The whole datatype can be found in listing 3.1.

The datatype storing all registered functions is called `defs`. It is a simple function mapping a function id to an expression wrapped by an option. The option is again to deal with possible namespace collisions later.

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

For primitive functions, there is no mapper but a direct primitive application function. It takes the name of a primitive function and a list of arguments. The application function should be universal in order to support any kinds of primitive functions on different machines. As the translation schema demands addition, we assume the function `sum` to be defined. All the arguments need to be of the defined number type. We can assume this function to be constant, although it takes an arbitrary amount of arguments. This choice is justified 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"
```

Inside a function, the passed arguments will be available as local variables. Those local variables are contained in the environment, represented by a list of values. We can refer to a value inside an expression through `Ident`. The selected number represents the index of the environment list.

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

The evaluation semantic of the form

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

```
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

can be read as *The expression e evaluates under the local variables ρ and the definitions ϕ to the value v*. The corresponding semantic is given in figure 3.2.

For the eval semantic, we can show determinism stated by

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

The proof is an induction over eval, which can be solved mostly automatically by metis and blast. In the next step, we define a step-counting semantic of the form

$$\rho,\phi \vdash e \rightarrow\mathrm{s}\ (v,t)$$

which is read as *The expression e evaluates under the given local variables ρ and the definitions ϕ to the value v involving t non-primitive function applications*. The corresponding semantic is given in figure 3.3.

We now want to show the equivalence of the semantics according to the resulting value. Showing the implication from the step-counting semantic to the normal semantic can be done by induction over the induction schema of the step-counting semantic. The auto tactic can handle all cases.

```
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
```

```
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

For the other direction, we need a lengthy auxiliary lemmo to deal with the arguments of function applications. It 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)))"
```

The lemma is shown as an induction over the length of vs with arbitrary es using the assumptions. While the simp tactic can handle the Nil case, the Cons case needs more effort. We get the assumption tex as premise for v#vs. We can easily show that this implies the same for vs. With this result, we can now use the IH and obtain the ts wanted for the vs. We can obtain t for v from the assumption of tex. Combining this gives the second part of the result. The length equality can be seen easily from here, which completes the proof.

With the help of this lemma it is now possible to show the direction from eval semantic to step-counting semantic by induction of the eval semantic. These results give us the equivalence of the two semantics according to the evaluation 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. This setup is quite laborious in Isabelle while only providing a proof for an easy-to-see step. Therefore, we neglect this proof.

We can now define the main function, which converts an expression to its cost function. The used schema corresponds to the one presented in 2.1.

```
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)"
```

Listing 3.4: Translation function for expressions

Now, we add a definition to convert a function expression. It uses the $\mathcal{T}$ function and adds 1+ to the front representing the function call. We also define the function conv based on this function. It alters a definition and registers the cost function for each defined function. Already defined cost functions will be dropped.

```
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 theorem, we need a property claiming that no timing function is defined in a given definition. This property avoids errors of already referenced timing functions being dropped by the conv function.

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

In order to better deal with this property, we define another auxiliary lemma. It states that if there is no timing function, then the evaluation result does not change when the expression is converted using conv. The lemma is proved over induction with a helping lemma claiming 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 theorem. Given an expression and a definition without a timing function. If an evaluation of the step-counting semantics results in `t`, then the evaluation of the converted expression and function definitions results in 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 is proved by induction over the step-counting semantic. While the simplest cases, Id and C, can be solved automatically, we need an Isar proof for the other cases. In the following, we look at the most important case F, which represents the application of a function. We have the following assumptions:

$$\text{length es} = \text{length vs} = \text{length ts} \tag{3.1}$$

$$\forall i < \text{length vs.}\ (\rho, \phi \vdash \text{es ! i} \rightarrow_s \text{es ! i} \rightarrow \text{N (ts ! i)} \wedge \tag{3.2}$$
$$(\text{no\_time}\phi \longrightarrow \rho, \text{conv}\phi \vdash \mathcal{T}(\text{es ! i}) \rightarrow \text{N (ts ! i)}))$$

$$\phi\ \text{f} = \text{Some fe} \tag{3.3}$$

$$\text{vs}, \phi \vdash \text{fe} \rightarrow \text{(v,t)} \tag{3.4}$$

$$\text{no\_time}\ \phi \tag{3.5}$$

$$\text{vs, conv}\ \phi \vdash \mathcal{T}\ \text{fe} \rightarrow \text{N t} \tag{3.6}$$

3.6 is already collapsed as the premise is given by 3.5. We need to show

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

The cost for a function application consists of the cost of the function evaluation and the cost of evaluating the arguments. We start with the cost for the function evaluation represented by evaluating the timing 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 3.6 and the axiom for primitive functions. From here, we only need to use the definition of the cost function to gain a statement about the cost term of the function.

$$\text{vs, conv}\ \phi \vdash \text{cost fe} \rightarrow \text{N (1 + t)}. \tag{3.7}$$

In the next step, we show that calling the cost function results in the same as evaluating its expression. For this, we need the evaluation of the arguments `es`. The evaluated result `vs` is already given for the step-counting evaluation in 3.2. Here, we can use the lemma `eval_count_eval` stating that the result is the same for the normal semantic.

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

From this, we use the lemma eval_no_time_trans with 3.5 and can conclude that the result stays the same when the function definitions are converted.

$$\forall \text{ i<length vs. } \rho, \text{conv } \phi \vdash \text{es ! i} \rightarrow \text{vs ! i} \tag{3.8}$$

From 3.3 and 3.5, we can conclude that f is an identifier for timing functions. Therefore

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

holds. Using this with 3.1 and the equation 3.8, we can use the rule for function application to gain the wanted intermediate step

$$\rho, \text{conv } \phi \vdash \text{cFun fn\$ es} \rightarrow \text{N (1 + t).} \tag{3.9}$$

The next step involves combining 3.9 into one list with the cost evaluation of the arguments. From 3.2 with 3.1 and 3.5 the mapping of the arguments to their cost can be deducted.

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

Combined with equation 3.9, we get the full list of arguments for the final 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, 3.1 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 the function application case.

4 Implementation

The main work of this thesis was the implementation of an automatic converter of functions to their running time function for Isabelle. In order to keep the converter readable, it is restricted to non-curried functions. The implementation works for the newest developer version of Isabelle and will be part of the Isabelle distribution.

This chapter explains the usage and the behavior of the automatic converter. Section 4.1 shows the form of the added commands. In the next section 4.2, we look at the conversion schema used by the converter. In order to provide a pleasing user experience, an automatic termination proof exists for the running time function. The details can be found in section 4.3. Defining running time functions can get complicated, so some restrictions exist to keep the implementation readable. Section 4.4 describes those restrictions.

4.1 Commands

The implementation provides three commands. The main one is `define_time_fun`. It accepts the name of a function as argument, converts it and registers its running time function. Additionally it tries to proof the termination automatically. Details on this proof can be found in section 4.3.

Although the termination proof should work automatically for most functions there are still edge cases, where it takes up too much time or even fails. Therefore the command `define_time_function` exists. Similar to the command `function` it only registeres the running time function. Afterwards the user needs to prove termination manually using the command `termination`.

In order to convert mutual recursive functions, the name of all the related functions need to be provided. This cannot be used to convert multiple not related functions at once. Sometimes we also want to specify the used function equations explicitely. This can be helpful for some cases as explained in 4.4.2. To add the equations one needs to use the keyword `equations` followed by the wanted equations. The full command schema can be found in listing 4.1.

The translation schema converts functions marked as zero differently, see section 4.2. With the command `define_time_0` more functions can me marked as zero. This should

```
define_time_fun {NameOfFunction}+ [equations {thm}+]
define_time_function {NameOfFunction}+ [equations {thm}+]
define_time_0 {NameOfFunction}
```

Listing 4.1: Schema of implemented command

```
+, -, *, /, div, <, ≤, Not, ∧, ∨, =, Num.numeral_class.numeral
```

Listing 4.2: Zero functions by default

be used with care and only for functions with a constant running time. Section 4.4.2 includes a discussion on that. In listing 4.2 all default zero functions can be found.

By default timing functions will be registered with the prefix "`T_`". To change this behaviour the configuration variable `time_prefix` can be adjusted. Most of the time this should be not be changed in order avoid confusion and incompatibilities.

The converter will try to register the function without sequential mode first, which is the default of the `function` command. This behaviour is chosen as it does not change the given equations. Therefore it avoids differences between the original and the timing function, which would create problems for the automatic termination proof in some cases. This will fail in some cases as for incomplete matching. Then the converter falls back to sequential mode and prints out a warning.

4.2 Schema

For first-order functions, the translation schema equals the schema used by Nipkow [Nip23]. All cases defined by Sands also equal this schema [San90]. Therefore, the proof described in chapter 3 holds for this restricted part of the schema.

Just like Nipkow, we treat some functions differently. Here, those functions are called zero functions. They include constructors and some basic mathematical operations and comparisons. Additionally, the user can mark any function as zero function. The command and the functions marked as zero by default can be found in chapter 4.1. Only functions taking a constant amount of time should be marked as zero functions. The user is obliged to mark only correct functions. Section 4.4.2 contains a discussion about this.

Additionally, the schema was extended for higher-order functions similar to Sands as described in chapter 2.2. Every argument being a function will be replaced by a pair of the function and its timing function. We first need to define two helper functions

$$
\begin{aligned}
\mathcal{N}[\![f]\!] &= (\texttt{fun}\ f) && \mid \text{passed function}\\
\mathcal{N}[\![e_1\ \$\ e_n]\!] &= \mathcal{N}[\![e_1]\!]\ \$\ \mathcal{N}[\![e_2]\!] && \mid \text{other expressions}
\end{aligned}
$$

Figure 4.1: Handling function application for normal evaluation

$$
\begin{aligned}
\mathcal{A}[\![f]\!] &= (f,\ T\_f) && \mid \text{identifier of a defined non-primitive function}\\
\mathcal{A}[\![f]\!] &= (f, (\lambda x\ \ldots\ z.\,0)) && \mid \text{identifier of a defined primitive function}\\
\mathcal{A}[\![f]\!] &= f && \mid \text{passed function}\\
\mathcal{A}[\![e]\!] &= \mathcal{N}[\![e]\!] && \mid \text{other expressions}
\end{aligned}
$$

Figure 4.2: Preparing arguments for timing functions

to deal with those constructs. $\mathcal{N}$ will replace all occurences of a passed function f by `(fst f)`. As a result, we get an expression evaluating to the normal result as in the original function. The definition can be found in figure 4.1. The function $\mathcal{A}$ converts an expression for being passed as an argument to a timing function. All defined constants of non-primitive functions will be converted into the wanted pair. For primitive functions, the second argument will be a lambda, taking the same number of arguments and returning 0. We do not need to change something for passed functions, as they are already pairs. All the other expressions will be converted by the function $\mathcal{N}$ as they should be evaluated as normal. As we do not support curried functions, those expressions cannot evaluate to a function and, therefore, will not create problems. The definition for $\mathcal{A}$ is described in figure 4.2.

We now define the function $\mathcal{F}$, which converts a given function application into the application of the timing function. All zero functions will be translated to 0 as their evaluation does not cost anything by definition. Defined functions get translated to the application of their timing function. All functions given as arguments should also be translated to an application of their timing function. As those arguments are now represented as a pair, we need to use the second element to receive the timing function. We use the previously defined function $\mathcal{A}$ to prepare arguments for the timing function. We will handle the cost of the arguments in the next step. The schema is defined in figure 4.3.

$\mathcal{T}$ is the main conversion function, defined in figure 4.4. It converts expressions just as defined by Nipkow. The only exceptions are expressions that need to be evaluated normally. As the schema provided by Nipkow is restricted to first-order functions, we

$$
\begin{aligned}
&\mathcal{F}[\![f\ a_1\ \dots\ a_n]\!] = 0 && \mid \text{Zero function}\\
&\mathcal{F}[\![f\ a_1\ \dots\ a_n]\!] = (T\_f\ \mathcal{A}[\![a_1]\!]\ \dots\ \mathcal{A}[\![a_n]\!]) && \mid \text{Defined function}\\
&\mathcal{F}[\![f\ a_1\ \dots\ a_n]\!] = ((snd\ f)\ \mathcal{A}[\![a_1]\!]\ \dots\ \mathcal{A}[\![a_n]\!]) && \mid \text{Passed function}
\end{aligned}
$$

Figure 4.3: Handling function applications

$$
\begin{aligned}
&\mathcal{T}[\![c]\!] && = 0\\
&\mathcal{T}[\![f\ a_1\ \dots\ a_n]\!] && = \mathcal{F}[\![T\_f\ a_1\ \dots\ a_n]\!] + \mathcal{T}[\![a_1]\!] + \cdots + \mathcal{T}[\![a_n]\!]\\
&\mathcal{T}[\![\text{if } c \text{ then } et \text{ else } ef]\!] && = \mathcal{T}[\![c]\!] + (\text{if } \mathcal{N}[\![c]\!] \text{ then } \mathcal{T}[\![et]\!] \text{ else } \mathcal{T}[\![ef]\!])\\
&\mathcal{T}[\![\text{case } e \text{ of } c_1 \Rightarrow e_1 \mid \dots \mid c_n \Rightarrow e_n]\!] && = \mathcal{T}[\![c]\!] +\\
& && \quad (\text{case } \mathcal{N}[\![e]\!] \text{ of } c_1 \Rightarrow \mathcal{T}[\![e_1]\!] \mid \dots \mid c_n \Rightarrow \mathcal{T}[\![e_n]\!])\\
&\mathcal{T}[\![\text{let } x = e_1 \text{ in } e_2]\!] && = \mathcal{T}[\![e_1]\!] + (\text{let } x = \mathcal{N}[\![e_1]\!] \text{ in } \mathcal{T}[\![e_2]\!])
\end{aligned}
$$

Figure 4.4: Main conversion schema for expressions

need to pass those expressions through our defined function $\mathcal{N}$. This happens in the cases of `if-else`, `case` and `let`.

Finally, we can define the function $\mathcal{C}$ transforming function definitions. The converter differs between recursive and non-recursive functions. Recursive functions will be translated with a leading $1+$, while this is left out at non-recursive functions. This can be justified as the function call only represents a constant at non-recursive functions. Therefore, the asymptotic running time class does not change. The schema is defined in figure 4.5.

$$
\begin{aligned}
&\mathcal{C}[\![f\ a_1\ \dots\ a_n = e]\!] = (T\_f\ a_1\ \dots\ a_n = \mathcal{T}[\![e]\!]) && \mid \text{non-recursive}\\
&\mathcal{C}[\![f\ a_1\ \dots\ a_n = e]\!] = (T\_f\ a_1\ \dots\ a_n = 1 + \mathcal{T}[\![e]\!]) && \mid \text{recursive}
\end{aligned}
$$

Figure 4.5: Conversion of function definitions

4.3 Termination proof

The command `define_time_fun` tries to automatically prove termination of the timing function. Therefore, it uses two different tactics. The first try equals the command `fun` as the command name suggests. Both use the tactic `lexicographic_order` in order to prove termination. We now look at the following function `sum`, where this tactic fails.

```
function sum :: "nat ⇒ nat ⇒ nat" where
  "sum i j = (if j ≤ i then 0 else i + sum (Suc i) j)"
  by pat_completeness auto
termination
  by (relation "measure (λ(i,j). j - i)") auto
```

Termination needs to be proved manually. Therefore, the first tactic also fails for the timing function. However, as we have already proved termination for this function, we can use it for the running time function. The second strategy does this and tries to cover all functions. In the first step, we register the timing function equivalent to using the `function` command.

```
function (domintros) T_sum :: "nat ⇒ nat ⇒ nat" where
  "T_sum i j = 1 + (if j ≤ i then 0 else T_sum (Suc i) j)"
  by pat_completeness auto
```

Listing 4.3: Function registration

In Isabelle, every function needs to terminate. Before this, the simp rules are not usable. However, we receive another function called *T_sum_dom*. It represents the domain of arguments in which *T_sum* terminates. Therefore, it takes the arguments of *T_sum* as a tuple and yields *True* if the function terminates for them and *False* otherwise. Based on this, the rules psimps are generated. They state the following: Under the assumption of *T_sum* terminating, the corresponding simp rule holds. The psimps rule for *T_sum* is given in equation 4.1.

$$\begin{aligned}&\texttt{T\_sum\_dom (?i, ?j)}\\ &\Longrightarrow \texttt{T\_sum ?i ?j = 1 + (if ?j} \leq \texttt{?i then 0 else T\_sum (Suc ?i) ?j)}\end{aligned} \tag{4.1}$$

From this equation, we can see how the termination proof works. In order to obtain the simp rules, we need to show that *T_sum_dom* holds for every argument. Before we start with the proof, we need to look at another set of generated rules. The `domintros` rules state when a function call terminates. Termination happens if all the recursive calls made also terminate. Those rules are not generated by default due to performance reasons. We needed to explicitly pass the option domintros to obtain them. This already happened in the listing 4.3. For our sum function, the domintros rule has the following form shown in equation 4.2. From this, we can see that a function call with the variables `i` and `j` with `j > i` terminates if the function call with `Suc i` and `j` terminates.

$$(\neg\ ?j \leq ?i \Longrightarrow \texttt{T\_sum\_dom (Suc ?i, ?j)}) \Longrightarrow \texttt{T\_sum\_dom (?i, ?j)} \tag{4.2}$$

This gives us all the rules we need to prove our goal. We start by setting up the goal of the form “$\texttt{T\_f\_dom (a}_1\texttt{,...,a}_n\texttt{)}$”. On this goal we perform an induction with the induction schema provided by the original function. This is already the step where we use the termination proof of the original function, as the induction schema is proved through the termination. To argue about the next step, we need to look at the translation schema for our timing functions. Taking the if-else construct as an example, the place where recursive function calls are made does not change. All function calls inside the condition will still be executed without another precondition. For the function calls inside the then and else branches, the preconditions stay the same, as the condition is evaluated as in the original function. This justifies why the resulting cases stay close to the original function. Taking the sum function as an example, the induction creates the goal

$$\bigwedge \texttt{i j.}\ \ (\neg\ \texttt{j} \leq \texttt{i} \Longrightarrow \texttt{T\_sum\_dom (Suc i, j)}) \Longrightarrow \texttt{T\_sum\_dom (i,j)}.$$

As expected, it is similar to the domintros rule shown in equation 4.2. With its help, we are also able to solve the goal. In order to support as many cases as possible, we use metis as an advanced prover. With the just-proven goal, the auto tactic can now prove termination. The whole proof can be found in listing 4.4.

```
lemma T_sum_dom: "T_sum_dom (i,j)"
  apply (induction i j rule: sum.induct)
  apply (metis T_sum.domintros)
  done
termination
  by (auto simp: T_sum_dom)
```

Listing 4.4: Proof schema over dom with help of original function

Internally, auto is used before metis, as it can do some more simplifications and, therefore, cover more edgecases. For functions with multiple equations, the induction schema will create multiple goals. The automation first tries to solve every goal by the corresponding domintros rule and falls back to all domintros rules in case of failure. This behavior reduces the number of “Unused theorems” warnings. The named lemma in listing 4.4 is just for demonstration purposes. The converter works with only internally usable goals.

4.4 Restrictions

Converting functions to their timing function starts simple if it is restricted to simple functions. However, as always, supporting more and more functions makes the converter's code bigger and more challenging to read. In order to keep the code maintainable, some restrictions were set. The following section is going to explain and justify the current restrictions.

4.4.1 Functions in datatypes

As described earlier, there is a translation for functions that are passed as arguments. Extending this for functions contained in pairs is straightforward, as the datatypes inside of pairs are not fixed. The converter supports this automatically. For arbitrary datatypes, this no longer holds. For example, imagine a datatype with a constructor taking a function. We cannot change the argument to a pair of functions as the datatype already fixes it. Therefore, creating a new datatype taking the mentioned type would be needed. As creating new datatypes is not in the wanted scope of this converter, it is not supported.

4.4.2 Operations on datatypes

Most basic operations, such as the equals operator "=", are marked as zero functions. We can easily argue that a comparison can be made in constant time for a simple datatype such as nat. This no longer holds for slightly more complicated types as lists. The exact time for comparing two lists would at least be linear through the length of the list. The command would need to register a timing version for every datatype the operator could be used with. In order to support all datatypes, this had to happen in the `datatype` command itself. Therefore, it is the user's responsibility to only use those zero operators in timing functions if the constant time can be justified. This would be the case for the equality operator if one of the sides is a constant as the empty list.

Also, the `length` function for lists is part of this problem. The function is an abbreviation of the size operator, which is automatically created for each datatype. Converting it directly will create problems. Instead, we can use size equations created for the list datatype. Specifying them in the command will give us the desired result. Listing 4.5 shows the full command.

4.4.3 Partial application

As described in chapter 2.3, Sands also proposes a translation schema for curried functions. This schema cannot be used here as Isabelle uses a strict type system. Trying

```
define_time_fun length equations list.size(3) list.size(4)
```

Listing 4.5: Converting the length function correctly

to define Sands' `app'` function as defined in listing 2.6 will fail in Isabelle. This is the case because, inside the then branch, the outcome of the func part is returned, while the else branch returns a triple. However, the function `capp` defined in listing 2.7 can be registered. However, as the arity counter is not coupled to the timing function itself, the timing function always needs to be of the form `'a ⇒ nat`. This type is not a wanted behavior, as we also want to evaluate the cost of a function with more than one argument left.

To overcome this issue, we would need to couple the counter to the number of arguments more closely. As this would involve a more complex datatype, this is outside the scope of the converter.

4.5 Probably examples

5 Summary

This work tries to automate the conversion of functions to their running time function in Isabelle. Therefore we look at existing conversion schemas. Naturally we start by restricting the function in the beginning and look at Non-Curried First Order Functions. The conversion schema for this functions are quite common and similar in differnet papers. For Higher Order Functions only Sands explains an extension for the existing schema. He extends the given function to a tuple containing the function and the timing function. In order to also deal with curried functions he extends this pair by a counter for the number of arguments left. In the end we gain a schema able to deal with Higher Order Curried functions. Sands provides prooves for all this schemas. Here we only take a look at the correctness proof of the schema for First Order Functions. The work provides a formalization of it in Isabelle and explains it.

In the main part an automatic converter for functions into their running time function in Isabelle is provided. As the schema for Curried Function gets too complicated we restrict ourself to Non-Curried Higher Order Functions. Additionally we don't allow functions to be passed in datatypes as conversion of it is no quite obvious and get laborious. The used schema is based on the proposals by Sands [San90] and Nipkow [Nip23]. As a result we gain a simple command able convert the restricted functions automatically. It contains an automation for the termination proof based based on the `lexicographic_order` tactic first. If this fails a more advanced schema is used to proof termination based on the termination of the original function. This proof should covers most functions. The command was added to the Isabelle source files and will be accessible in the next major verison.

The schema for curried function proposed by Sands cannot be used in Isabelle. Therefore it is needed to think about an adaption of this schema to overcome the current limitation on non-curried functions. Additionally there is a restrictions for functions as equality defined for and by every datatype. Here we would need to provide a running time version specific for every datatype not just function. As conversion is not hard once the wanted equations have been found, extending this is a rather technical task.

List of Figures

2.1 Translation function $\mathcal{T}$ for expressions of first-order functions 3
2.2 Translation function $\mathcal{C}$ for function definitions of first-order functions . 3
2.3 Translation schema for higher-order function with expression e 4
2.4 Translation schema V for higher-order functions 5
2.5 Translation T for higher-order functions 5
2.6 Converting application in case of curried higher-order functions 7

4.1 Handling function application for normal evaluation 18
4.2 Preparing arguments for timing functions 18
4.3 Handling function applications 19
4.4 Main conversion schema for expressions 19
4.5 Conversion of function definitions 19

Bibliography

[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.