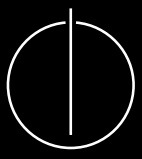
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Automatic Definition of Running Time Functions

Jonas Stahl



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

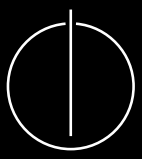
TECHNICAL UNIVERSITY OF MUNICH

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:	15.02.2024



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

Munich, 15.02.2024 Jonas Stahl

Abstract

Analyzing a function's running time is crucial in the analysis of algorithms. Timing functions are an essential step to measure the exact running time of a function. With their help, we can proof a certain running time class. In the proof assistant Isabelle, the conversion step of a function to their running time equivalent is done manually. This manual step introduces errors and makes following proofs worthless. This work aims to implement an automatic converter for functions to their timing functions in Isabelle.

In this work, we consider the number of function calls as a measurement of the running time of a function. We analyze Sands' and Nipkow's translation schemas for non-curried first-order functions. Sands provides an extension to support higher-order and curried functions. Implementing the schema for curried functions is not possible due to strict typing in Isabelle. Therefore, we restrict the converter's schema to non-curried higher-order functions. We formalize Sands' correctness proof for the translation schema for first-order functions.

The implemented converter simplifies the conversion of functions. It offers automatic translations through a single command and proves termination automatically. We use a tactic based on the termination proof of the original function. This tactic covers most functions. Manual termination is also possible in case of failure. We discuss the current restrictions of the converter, including functions within datatypes and dealing with functions without cost.

Contents

1 Introduction

The running time class is one of the most important measures to examine new algorithms and data structures. In order to argue about it, we first need to convert the functions into their running time functions. They measure the running time of a function. Here, we use the number of function calls to do so. Afterward, we can use proof assistants such as Isabelle to proof a certain running time. In Isabelle, the user needs to do this conversion step manually. This can create errors and makes the following proof worthless. This work aims to automate the conversion from functions to their running time functions in Isabelle. As we only consider the running time class relevant, we simplify the resulting function, leading to simpler proofs. To do so, we will drop all constants that are not needed. Additionally, we restrict the convertible functions to keep the converter readable and maintainable.

Starting with Chapter 2, we look at the conversion schema used in literature. We start with a schema for more restricted functions and relax the restrictions till we see a schema for curried higher-order functions. To prove the correctness of the used schema, we will look at a proof for the conversion schema in Chapter 3. The proof was formalized in Isabelle. We present the implementation in Chapter 4. The chapter overviews the added commands and explains the termination proof used. Afterward, we discuss the restrictions the converter currently has.

The converter gets added to the Isabelle distribution. Therefore, the work is based on the newest Isabelle development version at the time of the work. The implementation can be found in the associated GitHub repository [Sta].

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 [Nip24]. 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 Figure 2.3.

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 2.1.

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

Listing 2.1: Example function

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

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

Figure 2.3: Translation schema for case- and let-expressions by Nipkow

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

Figure 2.4: Translation schema for higher-order function with expression e

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

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 2.2.

```
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 2.2: 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 2.4.

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 2.5. 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 2.6. 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 2.3, 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
```

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

Figure 2.5: Translation schema V for higher-order functions

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

Figure 2.6: Translation T for higher-order functions

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

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 2.3: 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 2.4.

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

Listing 2.4: 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 2.5.

```
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 2.5: 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 2.6.

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

Figure 2.7: Converting application in case of curried higher-order functions

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

Listing 2.6: 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 Figure 2.5. The function $\mathcal{T}$ also stays the same as in Figure 2.6 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 2.7 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 2.7.

```
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 2.7: 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 [Sta]. 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

$$\rho, \phi \vdash e \to v$$

```
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 Listing 3.2.

For the eval semantic, we can show determinism stated by

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

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

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

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 Listing 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 Section 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 that tells us whether a definition contains any timing function. This allows us to avoid dropping already referenced timing functions with the conv function.

```
definition 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 function application. We have the following assumptions:

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

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

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

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

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

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

(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

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

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

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

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. The source code and more examples is in the associated GitHub repository [Sta].

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 Subsection 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. Subsection 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 Translation Schema

The translation schema of the converter is based on the schemas presented for first-order functions in Section 2.1. We use the schema by Nipkow [Nip24] for first-order functions as he provides a translation for all the needed cases. All cases defined by Sands also equal this schema [San90] as they are equal to Nipkow's translation. Therefore, the proof described in Chapter 3 holds for first-order function translated by this schema. In order to support higher-order functions, we extend the schema as described in Section 2.2. Every argument being a function will be replaced by a pair of the function and its timing function. We disallow datatypes containing functions as parameters, see Section 4.4.

Just like Nipkow and Sands, we treat some functions differently. Here, those functions are called zero functions. They include constructors and some basic mathematical operations and comparisons. The user can mark additional function as zero function. Section 4.1 contains the corresponding command and a list of the functions marked as

$$\mathcal{C}[\![f\ a_1\ \ldots\ a_n = e]\!] = (T\_f\ a_1\ \ldots\ a_n = \mathcal{T}[\![e]\!]) \quad \text{| non-recursive}$$
$$\mathcal{C}[\![f\ a_1\ \ldots\ a_n = e]\!] = (T\_f\ a_1\ \ldots\ a_n = 1 + \mathcal{T}[\![e]\!]) \quad \text{| recursive}$$

Figure 4.1: Conversion of function definitions

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

Figure 4.2: Main conversion schema for expressions

zero by default. Only functions taking a constant amount of time should be marked as zero functions. The user is in charge to mark only the correct functions. Subsection 4.4.2 contains a discussion about this.

We first 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 $1+$ for the cost of the function call, while this is left out at non-recursive functions. This is justified as the function call only represents a constant at non-recursive functions. Therefore, the asymptotic running time class does not change. It uses the function $\mathcal{T}$ to convert its expression into the timing version. Figure 4.1 defines the schema for $\mathcal{C}$.

$\mathcal{T}$ is the main conversion function, defined in Figure 4.2. It converts expressions just as defined by Nipkow. The only exceptions are expressions that need to be evaluated normally. For first-order functions, those expressions could be passed unchanged. We use the function $\mathcal{N}$ to achieve this for higher-order functions. This happens in the cases of `if-else`, `case` and `let`. Some optimizations are made to keep the translated functions simple in the same cases. The `if-else` and the `case` expression are left out if all of their cases are empty. Only the condition respectively the evaluated expression will be converted. In the translation of the let expression, cases exist where the assigned variable is no longer used in the translated body. Then, we reduce the translation to this body.

The function $\mathcal{F}$ described in Figure 4.3 handles the cost for function applications. Ap-

$$\mathcal{F}[\![f\ a_1\ \ldots\ a_n]\!] = 0 \quad \text{| Zero function}$$
$$\mathcal{F}[\![f\ a_1\ \ldots\ a_n]\!] = (T\_f\ \mathcal{A}[\![a_1]\!]\ \ldots\ \mathcal{A}[\![a_n]\!]) \quad \text{| Defined function}$$
$$\mathcal{F}[\![f\ a_1\ \ldots\ a_n]\!] = ((snd\ f)\ \mathcal{A}[\![a_1]\!]\ \ldots\ \mathcal{A}[\![a_n]\!]) \quad \text{| Passed function}$$

Figure 4.3: Handling function applications

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

Figure 4.4: Preparing arguments for timing functions

plications to zero functions get translated to 0 as their evaluation does not cost anything by definition. Application to defined functions gets translated to the application of their timing function. This timing function needs to be defined. Otherwise, the converter will throw an error. All functions passed 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. In Sands' schema this step happens identically in the function described in Figure 2.6.

For first-order functions, the arguments for functions need to be passed unchanged. As we get functions as pairs for higher-order functions, we must prepare the arguments before passing them. Function $\mathcal{A}$ does this preparation. 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 already passed functions, as they are pairs. All the other expressions need to be evaluated as in the original function. Again, we use the function $\mathcal{N}$ to achieve this behavior. 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 4.4.

Lastly, we define the mentioned function $\mathcal{N}$. This function replaces all occurrences of 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 4.5.

The functions $\mathcal{A}$ and $\mathcal{N}$ improve Sands' function $\mathcal{V}$ described in Figure 2.5. Sands replaces all primitive and non-primitive function identifer by the described pair except

$$\begin{aligned}
\mathcal{N}[\![e_1\ e_2]\!] &= \mathcal{N}[\![e_1]\!]\ \mathcal{N}[\![e_2]\!] && \text{| function application}\\
\mathcal{N}[\![f]\!] &= (\texttt{fst}\ f) && \text{| passed function}\\
\mathcal{N}[\![c]\!] &= c && \text{| constants / variables}
\end{aligned}$$

Figure 4.5: Convert equation for normal evaluation

for direct applications. With this conversion, he first registers a helper function that evaluates as the original function but accepts the pairs instead of functions as arguments. Through our function $\mathcal{N}$, we do not pass pairs to non-timing functions but the correct function argument. We use $\mathcal{A}$ to pass functions as pairs only for timing functions. We, therefore, have the advantage of not having to register an additional helper function.

As an example, we look at the default map function defined in Listing 2.3. At first, we see that the function map is recursive. Therefore, the calling cost is 1. The first case is then straightforward as it only contains the constant of the empty list. In the second case, we must use all our defined functions to deal with the passed function f. The application onto the Cons function is free because Cons is a constructor. The translation is as follows.

$\mathcal{C}[\![$`map f [] = []`$]\!]$
$\equiv$ `T_map f [] = 1 +` $\mathcal{T}[\![$`[]`$]\!]$
$\equiv$ `T_map f [] = 1`

$\mathcal{C}[\![$`map f (x#xs) = Cons (f x) (map f x)`$]\!]$
$\equiv$ `T_map f (x#xs) = 1 +` $\mathcal{T}[\![$`Cons (f x) (map f x)`$]\!]$
$\equiv$ `T_map f (x#xs) = 1 +` $\mathcal{F}[\![$`Cons (f x) (map f x)`$]\!]$ `+` $\mathcal{T}[\![$`f x`$]\!]$ `+` $\mathcal{T}[\![$`map f xs`$]\!]$
$\equiv$ `T_map f (x#xs) = 1 +` $\mathcal{T}[\![$`f x`$]\!]$ `+` $\mathcal{T}[\![$`map f xs`$]\!]$
$\equiv$ `T_map f (x#xs) = 1 +` $\mathcal{F}[\![$`f x`$]\!]$ `+` $\mathcal{T}[\![$`x`$]\!]$ `+` $\mathcal{F}[\![$`map f xs`$]\!]$ `+` $\mathcal{T}[\![$`f`$]\!]$ `+` $\mathcal{T}[\![$`xs`$]\!]$
$\equiv$ `T_map f (x#xs) = 1 +` $\mathcal{F}[\![$`f x`$]\!]$ `+` $\mathcal{F}[\![$`map f xs`$]\!]$
$\equiv$ `T_map f (x#xs) = 1 + ((snd f)` $\mathcal{A}[\![$`x`$]\!]$`) + T_map` $\mathcal{A}[\![$`f`$]\!]$ $\mathcal{A}[\![$`xs`$]\!]$
$\equiv$ `T_map f (x#xs) = 1 + ((snd f)` $\mathcal{N}[\![$`x`$]\!]$`) + T_map` $\mathcal{N}[\![$`f`$]\!]$ $\mathcal{N}[\![$`xs`$]\!]$
$\equiv$ `T_map f (x#xs) = 1 + ((snd f) x) + T_map f xs`

The result of this translation is similar to the translation Nipkow gives but with the passed function part of the pair. The type of the timing function is

`('a ⇒ 'b) × ('a ⇒ nat) ⇒ 'a list ⇒ nat.`

In the next example, we translate the function `Suc_all` to see how a function is passed to a timing function. The function is defined in Listing 4.3.

```
fun Suc_all :: nat list ⇒ nat list where
  Suc_all xs = map Suc xs
```

Listing 4.3: Example function increasing every element in list of natural numbers

`Suc_all` is a non-recursive function, meaning the calling cost of it is free. The function `T_map` is already defined We only need to prepare the function `Suc` for passing. `Suc` is a constructor and, therefore, a zero function. As a result, we represent the timing function of it by a lambda returning 0. The translation takes place in the following steps.

$\mathcal{C}[\![$`Suc_all xs = map Suc xs`$]\!]$
$\equiv$ `T_Suc_all xs =` $\mathcal{T}[\![$`map Suc xs`$]\!]$
$\equiv$ `T_Suc_all xs =` $\mathcal{F}[\![$`map Suc xs`$]\!]$ `+` $\mathcal{T}[\![$`Suc`$]\!]$ `+` $\mathcal{T}[\![$`xs`$]\!]$
$\equiv$ `T_Suc_all xs = T_map` $\mathcal{A}[\![$`Suc`$]\!]$ $\mathcal{A}[\![$`xs`$]\!]$
$\equiv$ `T_Suc_all xs = T_map (Suc, (`λ`x. 0)) `$\mathcal{N}[\![$`xs`$]\!]$
$\equiv$ `T_Suc_all xs = T_map (Suc, (`λ`x. 0)) xs`

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 4.4: 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 (4.1).

```
T_sum_dom (?i, ?j)
⟹ T_sum ?i ?j = 1 + (if ?j ≤ ?i then 0 else T_sum (Suc ?i) ?j)
```
(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 Listing 4.4. For our sum function, the domintros rule has the following form shown in (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.

```
(¬ ?j ≤ ?i ⟹ T_sum_dom (Suc ?i, ?j)) ⟹ T_sum_dom (?i, ?j)
```
(4.2)

This gives us all the rules we need to prove our goal. We start by setting up the goal of the form "`T_f_dom (a`$_1$`, ... ,a`$_n$`)`". 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

```
⋀i j.  (¬ j ≤ i ⟹ T_sum_dom (Suc i, j)) ⟹ T_sum_dom (i,j).
```

As expected, it is similar to the domintros rule shown in (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 4.5.

```
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 4.5: 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 4.5 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.

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

Listing 4.6: Converting the length function correctly

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 4.6 shows the full command.

4.4.3 Partial application

As described in Section 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 to define Sands' `app` function as defined in Listing 2.5 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 2.6 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 Examples

This chapter presents the translation of some interesting functions. Every listing first contains the original function, followed by the conversion command and the resulting timing function. The function `dummy` represents an arbitrary function. We use it in some examples to show more interesting edge cases.

The first example shows that a `if-else` expression is dropped if both cases are free of cost. Only the converted condition is left.

```
fun bool_to_nat :: bool ⇒ nat where
  bool_to_nat b = (if dummy b then 1 else 0)
define_time_fun bool_to_nat

fun T_bool_to_nat :: bool ⇒ nat where
  T_bool_to_nat b = T_dummy b
```

The same happens if we define the same function with a case construct.

```
fun bool_to_nat_case :: bool ⇒ nat where
  bool_to_nat_case b = (case dummy b of True ⇒ 1 | False ⇒ 0)
define_time_fun bool_to_nat_case

fun T_bool_to_nat_case :: bool ⇒ nat where
    T_bool_to_nat_case b = T_dummy b
```

A similar optimization happens for let expressions. In the following example, the converted function no longer uses the assigned variable of the inner `let` expression. Therefore, we reduce the expression to the body. The outer `let` is converted as expected.

```
fun let_add :: int ⇒ int ⇒ int where
  let_add x y = (let b = dummy y; a = dummy x in a + dummy b)
define_time_fun let_add

fun T_let_add :: int ⇒ int ⇒ nat where
  T_let_add x y = T_dummy y + (let b = dummy y in T_dummy x + T_dummy b)
```

If we want to convert a mutually recursive function, we need to provide the names of all related functions as specified in Section 4.1. All functions will be translated as recursive.

```
fun even :: nat ⇒ bool
  and odd :: nat ⇒ bool where
  even 0 = True
```

```
| odd 0 = False
| even (Suc n) = odd n
| odd (Suc n) = even n
define_time_fun even odd

fun T_even_T_odd :: nat ⇒ nat where
  T_even 0 = 1
| T_odd 0 = 1
| T_even (Suc n) = 1 + T_odd n
| T_odd (Suc n) = 1 + T_even n
```

The next example shows how powerful the termination proof is. First, we define the function calculating the collatz number. Sadly, we cannot prove the termination of this function, but we can assume it using sorry. Based on this assumption, the converter automatically proves the timing function's termination. Therefore, the command uses the proof over dom presented in Section 4.3. We receive a warning in the output.

```
function collatz :: nat ⇒ nat where
  collatz n = (if n ≤ 1 then 0 else
               if even n then 1 + collatz (n div 2)
               else 1 + collatz (3 * n + 1))
  by auto
termination sorry
define_time_0 "(dvd)"
define_time_fun collatz

fun T_collatz :: nat ⇒ nat where
T_collatz n = 1 + (if n ≤ 1 then 0 else
                   if even n then T_collatz (n div 2)
                   else T_collatz (3 * n + 1))
```

We now demonstrate how to use the `define_time_function` command. After applying it, we can prove termination using the `termination` command.

```
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
define_time_function sum
termination
```

```
  by (relation measure (λ(i,j). j - i)) auto

fun T_sum :: nat ⇒ nat ⇒ nat where
    T_sum i j = 1 + (if j ≤ i then 0 else T_sum (Suc i) j)
```

5 Summary

This work automates converting functions to their running time function in Isabelle. Therefore, we looked at existing conversion schemas by Sands [San90] and Nipkow [Nip24]. We presented a translation schema for non-curried first-order functions. Afterward, we extended this schema for higher-order functions. To deal with the functions as arguments, we extended them to pairs containing the function and the timing function. To support curried functions, we extended this pair by a counter for the number of arguments left. The next part looked at a correctness proof for the first-order functions schema. Sands presented this proof for his schema, and we formalized it in Isabelle.

The main part presents the automatic converter for functions into their running time function in Isabelle. We restricted the accepted functions to non-curried higher-order functions. This keeps the code readable and maintainable. Additionally, we restrict these functions' arguments. Their type may not be datatypes that contain other functions, as conversion is not obvious and quickly becomes laborious. The translation schema is based on the proposals by Sands [San90] and Nipkow [Nip24]. We can now use the command `define_time_fun` to convert a function automatically. It first tries to prove termination using the `lexicographic_order` tactic. If this fails, we use a more advanced tactic that uses the termination proof of the original function. This proof covers most functions. We can use the command `define_time_function` to prove termination manually.

Although the converter works automatically, we need to use it with care. Most basic mathematical operations and comparisons are marked as zero functions. Those zero functions will assumed as constant and get translated to 0. The constant time may not hold in certain situations as equality on lists. We would need to extend the datatype command to create a timing function for every overload. Also, curried functions are not allowed. Sands proposed a schema to tackle these, but it does not work in Isabelle due to the strict type system. We would need to consider adapting this schema to overcome this limitation.

The converter was added to the Isabelle distribution and will be accessible in Isabelle2024.

List of Figures

Bibliography

[Nip24] T. Nipkow. "Functional Data Structures and Algorithms A Proof Assistant Approach." In: (2024). [Online; accessed 01-February-2024].

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

[Sta] J. Stahl. *Timing function implementation and formalized proof*. `https://github.com/SecretJ12/BachelorThesisTimingFunctions`. [Online; accessed 01-February-2024].