hsrm Hochschule RheinMain

Programming Language Foundations

Winter term 2025/26

Prof. Dr. David Sabel

Last update: January 14, 2026

Theoretische Informatik
Fachbereich DCSM
Hochschule RheinMain
Unter den Eichen 5
65195 Wiesbaden

Email: david.sabel@hs-rm.de

Contents

1. Introduction

The course on "Foundations of Programming Languages" introduces the formal foundations of programming languages and the techniques and methods involved. The learning objective is to be able to apply most of these techniques.

Rather than dealing with syntax-based problems, such as lexing and parsing programs, the course is mainly concerned with the meaning of programs. The main question, therefore, is, how to define, represent, and reason about the meaning of programs. Before such a semantics of the language can be defined, however, the syntax has to be specified (which we will mostly do by using context-free grammars and additional side conditions). In order to specify the semantics, we will learn and examine the connections between several methods.

However, before we consider the semantics, we need to think about the language: Which programming language should we consider?

When classifying programming languages, several characteristics can be considered.

Programming Paradigms. There are two main classes of programming paradigms (and there also exist languages that combine them, such as Scala which is a multiparadigm language):

- Imperative programming languages: Languages that focus on *how* to execute tasks (e.g., C, C++, Python, Java). Object-oriented languages (C++,Java, etc.) are usually classified as a subclass of the imperative languages.
- Declarative programming languages: Languages that focus on *what* the program is supposed to compute. The main subclasses of declarative programming languages are logical programming languages (e.g. Prolog) and functional programming languages (e.g. Haskell, ML).

Level of Abstraction. Another classification is the level of abstraction:

- Machine languages: Low level languages closely associated with hardware (e.g.assembly).
- High-level languages: Languages that provide greater abstraction from the hardware (e.g., Haskell, Python, Java).
- Mid-level Languages: Languages that combine high-level abstractions with low-level capabilities (e.g., C).

Scope of Languages. It is also possible to classify languages according to their scope:

- General-purpose languages: Languages designed to be used for a wide range of programming tasks (e.g. Haskell, Python, C, Java, . . .).
- Domain-specific languages: Languages that are tailored to specific application domains and offer specialised features to facilitate tasks within that context (e.g. MATLAB for mathematical computations).

Computational Power We also can consider the computational power of languages:

- Turing completeness: Languages that can perform any computation that can be described algorithmically (e.g., JavaScript, C).
- Non Turing complete languages: Languages that are limited in their computational capabilities (e.g., simply typed lambda calculus, some domain-specific languages, etc.).

However, all modern programming languages have a rich syntax and thus lead to a very complex semantics, which means that they are out of scope for explaining the basic concepts of semantics and also for reasoning about them (there are research projects doing this for full languages).

Therefore, we will look for much more basic computational models, i.e. basic models that describe computation. These include

Turing Machine: Alan Turing's model of computation which will mainly be used to establish a notion of computability.

WHILE Language: A very simple imperative language that has variables, assignment, if-then-else, and while loops. However, its computational power is the same as that of Turing machines. So it can be considered as a core language of imperative languages. We will use the WHILE language to explain different styles of semantics.

Lambda Calculus: A model where functions and function applications are used to build programs. It can be seen as a core language of functional programming. Besides different evaluation strategies which model different categories of functional programming languages, we will discuss polymorphic typing after extending the calculus to make it more handy to program.

1.1. Outline

We start with a short introduction to computability, in particular we define Turing computability and recall the definition of a Turing machine in Chapter 2. The chapter ends with an explanation of the Church-Turing thesis. In Chapter 3 we will introduce the lambda calculus and different evaluation strategies for it. In Chapter 4 extensions of the lambda calculus are introduced, i.e. these languages are better suited as core languages of functional programming languages. In Chapter 5 the polymorphic typing of a functional language is explained and two algorithms for this static analysis are presented and analysed. In Chapter 6 different formalisms for defining formal semantics are explained and illustrated for an imperative core language. We conclude in Chapter 7. References and further reading on the various topics can be found in the corresponding chapters.

2. Computability

2.1. Intuitively Computable Functions

Every programmer has a sense of what a computer program can and cannot compute. However, it seems to be quite difficult to formalize a notion of "computable" and to prove that something can be computed or that something cannot be computed.

To give an intuition about computability, we first define a notion of computability that should coincide with the notion of "intuitively computable" (we cannot prove the coincidence, since "intuitively" cannot be captured formally). We then discuss the notion for some examples.

Definition 2.1.1 (Intuitive Computability)**.** *A function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ is called* computable *iff there exists an algorithm (a program in a modern programming language) that computes f, i.e. on input $(n_1, \ldots, n_k) \in \mathbb{N}_0^k$ the program terminates after a finite number of steps and returns $f(n_1, \ldots, n_k)$ as result. If f is a partial function, then for all inputs $(n_1, \ldots, n_k)$ for which is f is undefined, the algorithm should not terminate, but run forever.*

Example 2.1.2. *The algorithm with input $n \in \mathbb{N}_0$ and code*

```
while true {skip};
```

computes the partial function $f_1 : \mathbb{N}_0 \to \mathbb{N}_0$ which is undefined for all inputs (often written as $f_1(x) = \bot$).

The algorithm with inputs $n_1, n_2 \in \mathbb{N}_0$ and code:

```
result := n1 + n2;
return result;
```

computes the function $f_2 : (\mathbb{N}_0 \times \mathbb{N}_0) \to \mathbb{N}_0$ with $f_2(x, y) = x + y$.

Example 2.1.3. *The function*

$$f_3(n) = \begin{cases} 1, & \text{if } n \text{ is a prefix of the digits of the decimal representation of } \pi \\ 0, & \text{otherwise} \end{cases}$$

E.g. $f_3(31) = 1$ and $f_3(314) = 1$, but $f_3(2) = 0$ and $f_3(315) = 0$. The function f_3 is computable, since there are algorithms that can compute the first x digits of π, a subsequent comparison of the digits with the digits of n is also easily implementable.

Example 2.1.4. *Consider the function*

$$f_4(n) = \begin{cases} 1, & \text{if } n \text{ is a substring of the digits of the decimal representation of } \pi \\ 0, & \text{otherwise} \end{cases}$$

There seems to be no obvious algorithm, and thus it is not clear whether f_4 is computable. On the other hand, if we knew that π contains every sequence of numbers (an open problem), then the algorithm would be trivial by returning 1 on each input, and thus f_4 would definitely be computable.

Example 2.1.5. *For the function*

$$f_5(n) = \begin{cases} 1, & \text{if the digits of the decimal representation of } \pi \text{ contains the substring } 3^m \\ & \text{for some number } m \geq n. \\ 0, & \text{otherwise} \end{cases}$$

computability seems to be as hard as for f_4, but this is not true: Either there a fixed number m_0, such that π contains all words 3^m with $m \leq m_0$ and no words 3^m with $m > m_0$ or there is no such number. For the first case, the algorithm to compute f_5 is easy: if $n > m_0$, then return 0 else return 1. For the latter case $f_5(n) = 1$ holds for all n. So even though we do not know which of the two algorithms computes f_5, there exists an algorithm that computes f_5 and thus we know that f_5 is computable.

The last example shows that we only have to show existence of an algorithm that computes f, but that we do not need to construct the algorithm.

Example 2.1.6. *The function*

$$f_6(n) = \begin{cases} 1, & \text{if } P = NP \\ 0, & \text{if } P \neq NP \end{cases}$$

is computable, because either $P = NP$ holds (then $f_6(n) = 1$ for all n), or $(P \neq NP)$ holds (then $f_6(n) = 0$ for all n). Again, we do not know which algorithm is the right one, but this does not affect the computability of f_6.

Let f^r be the function and r be a real number

$$f^r(n) = \begin{cases} 1, & \text{if } n \text{ is prefix of the digits of the decimal representation of } r \\ 0, & \text{otherwise} \end{cases}$$

One might think, that f^r is computable for every real number r, since as we have argued f^π is computable. But there is a simple argument why this cannot be the case: The real numbers are not countable, but the algorithms (or programs of a programming language) are countable. Thus there are more functions f^r than algorithms and since one algorithm cannot be used for different numbers $r_1 \neq r_2$ (obviously there is prefix that distinguishes the digits of r_1 and r_2), there must be functions f^r that cannot be computed.

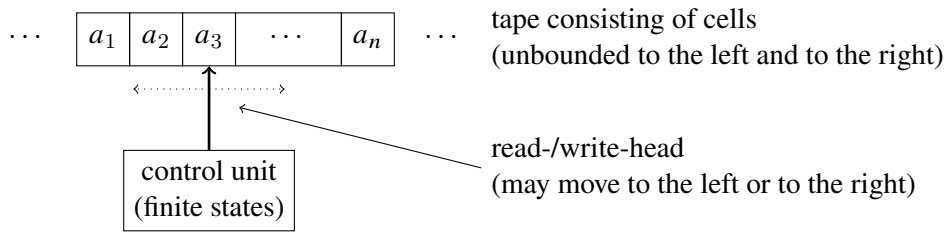


Figure 2.1.: Illustrating the Turing Machine

2.2. Turing Computability

We recall the Turing machine, which is a mathematical model for computation introduced in 1936 by the British computer scientist Alan Turing. An informal illustration of the Turing machine is given in Fig. 2.1. Turing machines use an infinite tape as memory. The tape is divided into cells. There is a read/write head which reads the content of the current cell (a symbol), the machine calculates the next state, and writes a new symbol into the current cell, finally it can move the head to the left or right direction by one cell.

The formal definition of Turing machines is as follows:

Definition 2.2.1 (Turing Machine)**.** *A* Turing machine *(*TM*) is a 7-tuple* $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ *where*

- Q *is a finite, non-empty set of* states*,*
- Σ *is a finite set of symbols, the* input alphabet*,*
- $\Gamma \supset \Sigma$ *is a finite set of symbols, the* tape alphabet*,*
- δ *is the* state transition function *where in the case of a* deterministic Turing machine *(DTM),* $\delta : (Q \times \Gamma) \to (Q \times \Gamma \times \{L, R, N\})$*, and in case of a* non-deterministic Turing machine *(NTM),* $\delta : (Q \times \Gamma) \to \mathcal{P}(Q \times \Gamma \times \{L, R, N\})$*,*
- $q_0 \in Q$ *is the* start state*,*
- $\square \in \Gamma \setminus \Sigma$ *is the* blank symbol*,*
- $F \subseteq Q$ *is the set of* final states*.*

For a deterministic Turing machine, an entry $\delta(q, a) = (q', b, x)$ means that in state q, if the content of the current cell is a, the next state will be q', the content of the current cell will be b and the read-/write-head moves into direction x (which is *left* if $x = L$, *right* if $x = R$ and no move if $x = N$).

For a non-deterministic Turing machine the same holds if $(q', b, x) \in \delta(q, a)$, but it means, that the TM can do this, but it can also do some other state transition in $\delta(q, a)$. It chooses non-deterministically between the choices in $\delta(q, a)$.

Definition 2.2.2 (A configuration of a Turing machine)**.** *A configuration of a Turing machine is a word* $wqw' \in \Gamma^* Q \Gamma^*$

A configuration wqw' means that the TM is in state q, the tape content is ww' and infinitely many blank symbols left and right from ww', and the current head position is on the first symbol of w'.

Initially the TM is in state q_0 and the head is on the first symbol of the input word.

Definition 2.2.3 (Start-configuration of a TM)**.** *For input w, the start-configuration of a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ is $q_0 w$.*

The transition relation $\vdash_M$ of a TM M is defined as a binary relation on configurations:

Definition 2.2.4 (Transition relation on configurations)**.** *Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be a TM. The relation $\vdash_M$ is defined by the following cases (where $\delta(q, a) = (q', c, x)$ in case of an NTM means $(q', c, x) \in \delta(q, a)$):*

1. *$wqw' \nvdash_M$ if $q \in F$ (no transition is possible for final states).*
2. *$b_1 \cdots b_m q a_1 \cdots a_n \vdash_M b_1 \cdots b_m q' c a_2 \cdots a_n$,*
   if $\delta(q, a_1) = (q', c, N)$, $m \geq 0, n \geq 1, q \notin F$
3. *$b_1 \cdots b_m q a_1 \cdots a_n \vdash_M b_1 \cdots b_{m-1} q' b_m c a_2 \cdots a_n$,*
   if $\delta(q, a_1) = (q', c, L)$, $m \geq 1, n \geq 1, q \notin F$
4. *$b_1 \cdots b_m q a_1 \cdots a_n \vdash_M b_1 \cdots b_m c a_2 q' a_3 \cdots a_n$,*
   if $\delta(q, a_1) = (q', c, R)$, $m \geq 0, n \geq 2, q \notin F$
5. *$b_1 \cdots b_m q a_1 \vdash_M b_1 \cdots b_m c q' \square$, if $\delta(q, a_1) = (q', c, R)$ and $m \geq 0, q \notin F$*
6. *$q a_1 \cdots a_n \vdash_M q' \square c a_2 \cdots a_n$, if $\delta(q, a_1) = (q', c, L)$ and $n \geq 1, q \notin F$*

Let $\vdash_M^i$ be the i-fold application of $\vdash_M$ and $\vdash_M^*$ the reflexive-transitive closure of $\vdash_M$. We omit the index M in $\vdash_M$ and write $\vdash$ is M is clear from the context.

We explain the different cases: Item 1 prohibits to proceed if the TM has reached a final state. Items 2 to 4 are standard cases where the read-write-head does not move, moves to the left, moves to the right. Item 5 covers the case, the TM has not discovered the symbols right from the current one, and now moves to the right. Then a new blank symbol is added to the configuration. Item 6 is the symmetric case for the left end of the tape.

Example 2.2.5. *The DTM $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, \square\}, \delta, q_0, \square, \{q_3\})$ with*

$$
\begin{array}{lll}
\delta(q_0, 0) = (q_0, 0, R) & \delta(q_0, 1) = (q_0, 1, R) & \delta(q_0, \square) = (q_1, \square, L) \\
\delta(q_1, 0) = (q_2, 1, L) & \delta(q_1, 1) = (q_1, 0, L) & \delta(q_1, \square) = (q_3, 1, N) \\
\delta(q_2, 0) = (q_2, 0, L) & \delta(q_2, 1) = (q_2, 1, L) & \delta(q_2, \square) = (q_3, \square, R) \\
\delta(q_3, 0) = (q_3, 0, N) & \delta(q_3, 1) = (q_3, 1, N) & \delta(q_3, \square) = (q_3, \square, N)
\end{array}
$$

interprets its input $w \in \{0, 1\}^*$ as binary number and add 1 to it. In start state q_0 the TM moves its head to the right end of input (until it detects the blank symbol $\square$), then it changes its state to q_1. In state q_1 it tries to add 1. If there is no carryover, it switches to state q_2 and moves its head to the left until it reaches the blank symbol and accept in state q_3. If there is a carryover of

1 *then it has to be added to the next digit. The DTM remains in state* q_1 *and moves its head to the left and so on. If the carryover remains also after reading the whole number (from right to left, for instance for input* 1111*), then an new digit is created by left from the first position by the transition* $\delta(q_1, \square) = (q_3, 1, N)$ *and the DTM accepts in* q_3*.*

As an example, wir consider the input 0011*. The execution of the TM is as follows:*

$q_0 0011 \vdash 0q_0 011 \vdash 00q_0 11 \vdash 001q_0 1 \vdash 0011q_0\square \vdash 001q_1 1\square \vdash 00q_1 10\square \vdash 0q_1 000\square \vdash q_2 0100\square \vdash q_2\square 0100\square \vdash \square q_3 0100\square$

While one can define the language accepted by a TM, we use the acceptance of a TM to define computability of functions on natural numbers, and also computability of functions on words over the input alphabet.

Definition 2.2.6. *Let* $bin(n)$ *be the binary representation of number* $n \in \mathbb{N}_0$*. A function* $f : \mathbb{N}_0^k \to \mathbb{N}_0$ *is called* Turing computable*, if there exists a DTM* $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ *such that for all* $n_1, \ldots, n_k, m \in \mathbb{N}_0$*:*

$$f(n_1, \ldots, n_k) = m \text{ iff } q_0 bin(n_1)\# \ldots \# bin(n_k) \vdash^* \square \ldots \square q_f bin(m)\square \ldots \square \text{ with } q_f \in F.$$

A function $f : \Sigma^* \to \Sigma^*$ *is called* Turing computable*, if there exists a DTM* $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ *such that for all* $u, v \in \Sigma^*$*:*

$$f(u) = v \text{ iff } q_0 u \vdash^* \square \ldots \square q_f v\square \ldots \square \text{ with } q_f \in F.$$

If $f(n_1, \ldots, n_k)$ is undefined (and f is a partial function), the TM may loop. Moreover, we can assume that this is always the case, by constructing an according TM from a given one.

Example 2.2.7. *The successor function* $f(x) = x + 1$ *for all* $x \in \mathbb{N}_0$ *is Turing computable. We defined the corresponding TM already in Example 2.2.5.*

Example 2.2.8. *The identity* $f(x) = x$ *for all* $x \in \mathbb{N}_0$ *ist Turing computable, since for DTM* $M = (\{q_0\}, \{0, 1, \#\}, \{0, 1, \#\square\}, \delta, q_0, \square, \{q_0\})$ *with* $\delta(q_0, a) = (q_0, a, N)$ *for all* $a \in \{0, 1, \#, \square\}$*, we have* $q_0 bin(n) \vdash^* q_0 bin(n)$ *for all* $n \in \mathbb{N}_0$*.*

The function $f(x) = \perp$ *which is undefined for every input is Turing computable, since* $M = (\{q_0\}, \{0, 1, \#\}, \{0, 1, \#, \square\}, \delta, q_0, \square, \emptyset)$ *with* $\delta(q_0, a) = (q_0, a, N)$ *loops for every input and never reaches a final state.*

Remark 2.2.9. *We also could have used multitape Turing machines which have* k*-tapes each of them with there own read-write-head. The notion of computability, however, remains the same, since single tape TMs can simulate multitape TMs (and vice versa).*

Let us recall the important result, that not all functions on natural numbers are computable. Turing machines and words can be encoded as numbers (usally called Gödel numbers).

Let f be a function that gets a number, and checks whether the number is a valid encoding of a tuple consisting of Turing machine M and a word w and returns 1 if the TM holds on this configuration and 0 otherwise.

Let f' be a function that gets a number, and checks whether the number is a valid encoding of a tuple consisting of Turing machine M and a word w and returns 1 if the TM holds on this configuration, and is undefined otherwise.

Then f is not Turing computable, because it is equivalent to solve the halting problem for Turing machines which is undecidable. The function f' is Turing computable, because f' can be computed by a Turing machine, by simulating M on word w.

2.3. The Church-Turing Thesis

Besides Turing and the definition of Turing computability, also other famous computer scientists and mathematicians have attempted to answer the question what can be computed and what cannot be computed. A lot of research has been done in the 1930s. Among them are Kurt Gödel and Jacques Herbrand (defining the class of general recursive functions) and Alonzo Church and Stephen Kleene (defining λ-definable functions). As a remarkable result it was shown, that all of the formalisms were shown to be equivalent, i.e. they define the same class of functions.

This lead to the Church-Turing thesis stating that the class of intuitively computable functions is the same as Turing computable functions.

> **Church-Turing Thesis:** The class of Turing computable functions is identical to the class of intuitively computable functions.

The thesis cannot be proved, since there is no formal definition of "intuitively computable".

The thesis however, is supported by showing Turing completeness of other formalisms:

Definition 2.3.1 (Turing completeness)**.** *A formalism (a programming language, a cellular automaton, an instruction set of a computer, a rewrite system etc.) is called* Turing complete *iff it can simulate a Turing machine. That means every Turing computable function can also be computed by the formalism.*

Surprisingly, a lot of of formalisms were shown to be Turing complete and thus they can be replaced by Turing computability in the Church-Turing thesis – since they all compute the same class of functions.

Among them are all modern programming languages, the lambda-definable functions, the general recursive functions, WHILE-programs, GOTO-programs, the RAM-model, etc. (the proofs can be found in introductionary books on computability theory).

You may convince yourself that your favourite programming language is Turing complete by programming a simulation of Turing machines.

2.4. Conclusion and References

Starting from a notion of intuitive computability, we recalled Turing machines and Turing computability. A lot of models are Turing complete and thus the power of Turing computability also holds for those models. The Church-Turing thesis states that all these notions of computability are exactly the intuitively computable functions. In conclusion, it makes sense to a consider foundational models and their semantics as a core of real programming languages, as long as the computability is not weakened in the models (or there is a good reason to do so). In the next chapter we will consider the Turing complete formalism of the (untyped) lambda calculus.

The presentation of intuitive computability including some examples and the presentation of Turing machines is oriented on (Schöning, 2008). Introductionary books covering Turing machines, computability and the Church-Turing thesis are for instance (Hopcroft et al., 2006; Sipser, 2013).

3. The Lambda Calculus

In this chapter we will introduce the untyped lambda calculus which is a foundational model of computation and the core of functional programming languages. However, lambda notation is used in other settings too. For instance, several non-functional programming languages like Java or Python have introduced functional concepts and lambda expressions. Also in mathematics, lambda notation is used to represent functions. We will also do this in other chapters, for instance, when introducing the denotational semantics of an imperative programming language.

However, our first goal is to introduce the syntax and the operational semantics of the lambda calculus with different variants that match non-strict functional programming languages like Haskell and strict functional languages like ML and F#.

For introducing the lambda calculus, we first present the syntax and then present different evaluation strategies which thus define different operational semantics.

3.1. Syntax of the Lambda Calculus

The lambda calculus was introduced by Alonzo Church in the 1930s (Church, 1941). The syntax is quite simplistic:

Definition 3.1.1. *Expressions of the lambda calculus are built according to the following grammar (starting with non-terminal* **Expr***):*

$$\textbf{Expr} ::= V \mid \lambda V.\textbf{Expr} \mid (\textbf{Expr}\ \textbf{Expr})$$

Here V is a non-terminal for variables (of a countable infinite set of variables, usually denoted as $x, y, z, \ldots$).

An expression of the form $\lambda x.s$ is called an *abstraction*. The lambda binder λx binds the variable x in the subexpression s (which is called the *body* of the abstraction), i.e. the *scope* of x is s. An abstraction represents an anonymous function, i.e. a function that does not have a name. For instance, the (non anonymous) identity function $id(x) = x$ is represented in the lambda calculus by the abstraction $\lambda x.x$.

The construct $(s\ t)$ is called an *application*. Applications allow applying functions to arguments. The expression in function position s and the argument t are arbitrary expressions. For example, the argument may again be an abstraction. That is why we also speak of the higher-order lambda calculus, since functions may have functions as arguments or as result. For example, one may apply the identity function to the identity function as $((\lambda x.x)\ (\lambda x.x))$

Convention 3.1.2. *To omit parentheses, we use the following associativities and priorities: Application is left-associative, i.e.* $s\ t\ r$ *means* $((s\ t)\ r)$ *and* not $(s\ (t\ r))$. *The body of an abstraction extends as far as possible. For instance,* $\lambda x.s\ t$ *means* $\lambda x.(s\ t)$ *and* not $((\lambda x.s)\ t)$. *As short-hand notation we write* $\lambda x_1, \ldots, x_n.t$ *for the nested abstractions* $\lambda x_1.(\lambda x_2. \ldots (\lambda x_n.t) \ldots)$.

Example 3.1.3. *Some prominent expressions of the lambda calculus are:*

$$\begin{array}{lcl} I & := & \lambda x.x \\ K & := & \lambda x.\lambda y.x \\ K_2 & := & \lambda x.\lambda y.y \\ \Omega & := & (\lambda x.(x\ x))\ (\lambda x.(x\ x)) \\ Y & := & \lambda f.(\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x))) \\ Z & := & \lambda f.(\lambda x.(f\ \lambda z.(x\ x)\ z))\ (\lambda x.(f\ \lambda z.(x\ x)\ z)) \\ S & := & \lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z) \end{array}$$

The expression I *is the identity function. The expressions* K *and* K_2 *take two arguments and* K *projects to the first one, while* K_2 *projects to the second one. The expression* Ω *is a non-terminating expression (we will see this later). The expression* Y *is a so-called fixpoint-combinator, which means that* $Y\ f$ *is (semantically) equal to* $f\ (Y\ f)$. *For* Y *this will hold for call-by-name evaluation, while for call-by-value evaluation,* Z *is a fixpoint-combinator. The expression* S *is the so-called S-combinator of the so-called SKI-calculus (which only has S, K and I).*

We define functions FV and BV that compute the sets of *free variables* and *bound variables* for a lambda expression.

$$\begin{array}{llll} FV(x) & = x & BV(x) & = \emptyset \\ FV(\lambda x.s) & = FV(s) \setminus \{x\} & BV(\lambda x.s) & = BV(s) \cup \{x\} \\ FV(s\ t) & = FV(s) \cup FV(t) & BV(s\ t) & = BV(s) \cup BV(t) \end{array}$$

Example 3.1.4. *For the expression* $s = (\lambda x.\lambda y.\lambda w.(x\ y\ z))\ x$, *computing the free and bound variables results in* $FV(s) = \{x, z\}$ *and* $BV(s) = \{x, y, w\}$.

If $FV(t) = \emptyset$, then we say that t is *closed* or a *program*, otherwise t is called *open*. An occurrence of a variable x in an expression s is called *bound* if it is in the scope of a binder λx, otherwise it is called *free*.

Example 3.1.5. *Let* $s = (\lambda x.\lambda y.\lambda w.(x\ y\ z))\ x$. *We label the occurrences of variables (not at the binders):* $(\lambda x.\lambda y.\lambda w.(\underbrace{x}_{1}\ \underbrace{y}_{2}\ \underbrace{z}_{3}))\ \underbrace{x}_{4}$. *The occurrence of* x *labeled with* 1 *and the occurrence of* y *labeled with* 2 *are bound occurrences. The occurrence of* z *labeled with* 3 *and the occurrence of* x *labeled with* 4 *are free occurrences.*

Exercise 3.1.6. *Let* $s = (\lambda y.(y\ x))\ (\lambda x.(x\ y))\ (\lambda z.(z\ x\ y))$. *Compute* $FV(s)$ *and* $BV(s)$. *Which occurrences of* x, y, z *are free, which are bound?*

Definition 3.1.7 (Capture-Avoiding Substitution)**.** *We write* $s[t/x]$ *for the expression* s *where all free occurrences of variable* x *are replaced by expression* t. *To avoid name and capturing conflicts, we assume that* $BV(s) \cap FV(t) = \emptyset$ *holds.*

With this assumption, substitution *can be defined inductively with the following equations:*

$$\begin{array}{lcl} x[t/x] & = & t \\ y[t/x] & = & y, \text{ if } x \neq y \\ (\lambda y.s)[t/x] & = & \begin{cases} \lambda y.(s[t/x]) & \text{if } x \neq y \\ \lambda y.s & \text{if } x = y \end{cases} \\ (s_1\ s_2)[t/x] & = & (s_1[t/x]\ s_2[t/x]) \\ (s_1\ s_2)[t/x] & = & (s_1[t/x]\ s_2[t/x]) \end{array}$$

For example, $(\lambda x.z\ x)[(\lambda y.y)/z]$ results in $(\lambda x.((\lambda y.y)\ x))$. Note that without the side condition, free variables could be captured by substitution, for instance, consider $(\lambda x.z\ x)[\lambda y.x/z]$: this would result in $\lambda x.((\lambda y.x)\ x)$ such that the free occurrence of x in $\lambda y.x$ would be captured (and thus become a bound occurrence). However, we forbid this case.

Exercise 3.1.8. *For* $s = (x\ z)\ (\lambda y.x)$ *and* $t = \lambda w.(w\ w)$, *compute* $s[t/x]$.

Contexts C are expressions where one subexpression is replaced by the context hole $[\cdot]$. Thus, they are constructed according to the following grammar with start symbol **Ctxt** (where **Expr** are generated as defined in Definition 3.1.1):

$$\textbf{Ctxt} ::= [\cdot] \mid \lambda V.\textbf{Ctxt} \mid (\textbf{Ctxt}\ \textbf{Expr}) \mid (\textbf{Expr}\ \textbf{Ctxt})$$

Replacing the hole of an expression by an expression results in a new expression. This kind of substituting the hole of C by expression s is denoted as $C[s]$ and it may capture variables of s. For instance, let $C = \lambda x.[\cdot]$, then $C[\lambda y.x]$ is the expression $\lambda x.(\lambda y.x)$. The free occurrence of x in $\lambda y.x$ is captured by the substitution.

3.2. α-Renaming and β-Reduction

Next we define α-renaming. A single α-renaming-step is of the form:

$$C[\lambda x.s] \xrightarrow{\alpha} C[\lambda y.s[y/x]] \text{ if } y \notin BV(C[\lambda x.s]) \cup FV(C[\lambda x.s])$$

The reflexive-transitive closure of $\xrightarrow{\alpha} \cup \xleftarrow{\alpha}$ is called *α-equivalence* and written $s =_\alpha t$. We do not distinguish α-equivalent expressions (and thus write $s = t$ also for α-equivalent expressions), but to avoid naming conflicts, we assume the following convention:

Definition 3.2.1 (Distinct Variable Convention (DVC))**.** *In any expression s, bound and free variables are disjoint, i.e. $BV(s) \cap FV(s) = \emptyset$, and all variables on binders are pairwise distinct.*

The convention can be obeyed by using α-renamings (use new variable names and rename all binders using α-renaming).

Example 3.2.2. *The expression $(y\ (\lambda y.((\lambda x.(x\ \lambda x.x))\ (x\ y))))$ violates the DVC, since x and y occur free and bound, and x occurs twice at a binder. We use α-renamings to satisfy the DVC:*

$$\begin{array}{ll} & (y\ (\lambda y.((\lambda x.(x\ \lambda x.x))\ (x\ y)))) \\ \xrightarrow{\alpha} & (y\ (\lambda y_1.((\lambda x.(x\ \lambda x.x))\ (x\ y_1)))) \\ \xrightarrow{\alpha} & (y\ (\lambda y_1.((\lambda x_1.(x_1\ \lambda x.x))\ (x\ y_1)))) \\ \xrightarrow{\alpha} & (y\ (\lambda y_1.((\lambda x_1.(x_1\ \lambda x_2.x_2))\ (x\ y_1)))) \end{array}$$

Exercise 3.2.3. *Let $s = ((\lambda x.(x\ \lambda y.(x\ z)\ (y\ x)))\ (\lambda z.y))$. Perform α-renamings for s, such that the resulting expression satisfies the DVC.*

Now substitution $s[t/x]$ can be performed in any case:

Definition 3.2.4 (Substitution)**.** *If $BV(s) \cap FV(t) = \emptyset$, then $s[t/x]$ is defined as in Definition 3.1.7. Otherwise, let $s' =_\alpha s$ such that s' fulfills the DVC. Then $BV(s') \cap FV(t) = \emptyset$ holds. Then let $s[t/x] = s'[t/x]$ using Definition 3.1.7 for $s'[t/x]$.*

The classical reduction rule of the lambda calculus is *β-reduction*: it evaluates the application of a function to an argument:

Definition 3.2.5. *The (direct) (β)-*reduction *is defined as*

$$(\beta) \qquad (\lambda x.s)\ t \xrightarrow{\beta} s[t/x]$$

If $r_1 \xrightarrow{\beta} r_2$, the we say r_1 directly reduces *to r_2.*
The contextual closure of β-reduction is $\xrightarrow{C,\beta}$ defined as

$$C[s] \xrightarrow{C,\beta} C[t] \text{ iff } C \text{ is a context and } s \xrightarrow{\beta} t.$$

Example 3.2.6. *Expression $(\lambda x.x)\ (\lambda y.y)$ can be β-reduced:*

$$(\lambda x.x)\ (\lambda y.y) \xrightarrow{\beta} x[(\lambda y.y)/x] = \lambda y.y$$

Expression $(\lambda y.y\ y\ y)\ (x\ z)$ can also be β-reduced:

$$(\lambda y.y\ y\ y)\ (x\ z) \xrightarrow{\beta} (y\ y\ y)[(x\ z)/y] = (x\ z)\ (x\ z)\ (x\ z)$$

To obey the DVC after a β-reduction, one has to rename the resulting expression: consider the expression $(\lambda x.(x\ x))\ (\lambda y.y)$. One β-reduction results in

$$(\lambda x.(x\ x))\ (\lambda y.y) \xrightarrow{\beta} (\lambda y.y)\ (\lambda y.y)$$

But $(\lambda y.y)\ (\lambda y.y)$ violates the DVC. With α-renaming we obtain $(\lambda y_1.y_1)\ (\lambda y_2.y_2)$, which satisfies the DVC.

For the evaluation of expressions, it is not sufficient to perform β-reductions on the top-level of expressions, since e.g. the expression $((\lambda x.x)\ (\lambda y.y))\ (\lambda z.z)$ could not be reduced. Thus $\xrightarrow{C,\beta}$-steps have to be used. However, as this is not deterministic, since for instance, the expression $((\lambda x.x\ x)((\lambda y.y)(\lambda z.z)))$ has two positions, where a β-reduction is possible (the corresponding subexpressions are called a *redex* (short form of reducible expression)):

- $\underline{((\lambda x.x\ x)((\lambda y.y)(\lambda z.z)))} \to ((\lambda y.y)(\lambda z.z))\ ((\lambda y.y)(\lambda z.z))$ or
- $((\lambda x.x\ x)\underline{((\lambda y.y)(\lambda z.z)))} \to ((\lambda x.x\ x)(\lambda z.z))$.

Fixing the position where to reduce is also called a *reduction strategy* and we will fix it, when defining the operational semantics of the lambda calculus. But before, we analyse *arbitrary* β-reductions at arbitrary positions.

3.3. Confluence and the Church-Rosser-Theorem

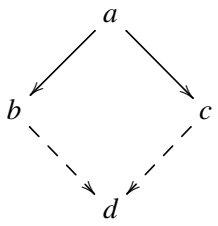
We define some notation: Let $\to \subseteq (M \times M)$ be a binary relation. We denote with

- $\leftrightarrow$ the symmetric closure of $\to$ (i.e. $a \leftrightarrow b$ iff $a \to b$ or $b \to a$).
- $\xrightarrow{i}$ the i-fold composition of $\to$ defined by $a \xrightarrow{0} a$ for all $a \in M$, and for $i > 0$: $a \xrightarrow{i} b$, if there exists $b' \in M$ such that $a \to b'$ and $b' \xrightarrow{i-1} b$.
- $\xrightarrow{i \vee j}$ is the union of the i-fold and the j-fold composition (i.e. $a \xrightarrow{i \vee j} b$ iff $a \xrightarrow{i} b$ or $a \xrightarrow{j} b$). In particular, $\xrightarrow{0 \vee 1}$ is the reflexive-closure of $\to$.
- $\xrightarrow{*}$ the reflexive-transitive closure of $\to$ (i.e. $a \xrightarrow{*} b$ iff there exists some $i \in \mathbb{N}_0$ such that $a \xrightarrow{i} b$).
- $\xleftrightarrow{*}$ the reflexive-transitive closure of $\leftrightarrow$.
- $\xrightarrow{+}$ the transitive closure of $\to$ (i.e. $a \xrightarrow{+} b$ iff there exists some $i \in \mathbb{N}$ such that $a \xrightarrow{i} b$).

The relation $\xleftrightarrow{C,\beta,*}$ is sometimes also called β-equivalence or also convertibility.

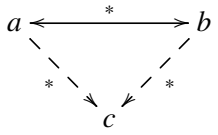
Definition 3.3.1. *A binary relation $\to \subseteq M \times M$ has the* diamond property *iff whenever $a \to b$*

and $a \to c$ there exists $d \in M$ such that $b \to d$ and $c \to d$.

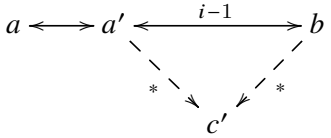


Relation $\to$ is confluent *iff $\overset{*}{\to}$ has the diamond property.*

Lemma 3.3.2. *If reduction relation $\to$ is confluent, then $a \overset{*}{\leftrightarrow} b$ implies $\exists c : a \overset{*}{\to} c \wedge b \overset{*}{\to} c$*

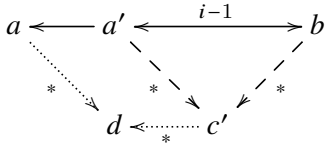


Proof. By induction of the given sequence $a \overset{i}{\leftrightarrow} b$. If $i = 0$, then $a = b$ and the claim is obvious. If $i > 0$, then there exists a' such that $a \leftrightarrow a' \overset{i-1}{\leftrightarrow} b$. The induction hypothesis gives us c' with $a' \overset{*}{\to} c'$ and $b \overset{*}{\to} c'$.



We distinguish two cases:

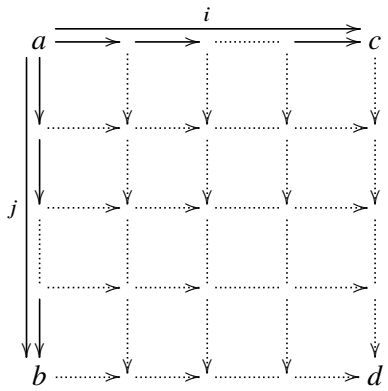
- $a \to a'$: Then $a \to a' \overset{*}{\to} c'$ and thus $a \overset{*}{\to} c'$. Since also $b \overset{*}{\to} c'$, the claim holds for $c = c'$.
- $a' \to a$: Then also $a' \overset{*}{\to} a$. Since $\to$ is confluent, $\overset{*}{\to}$ has the diamond property and thus from $a' \overset{*}{\to} a$ and $a' \overset{*}{\to} c'$, we obtain d with $a \overset{*}{\to} d$ and $c' \overset{*}{\to} d$. Since $b \overset{*}{\to} c' \overset{*}{\to} d$, the claim holds for $c = d$. We illustrate this case:



□

Lemma 3.3.3. *Let $\to$ be a binary relation and $\overset{*}{\to}$ be its reflexive-transitive closure. If $\to$ has the diamond property, then $\overset{*}{\to}$ has the diamond property.*

Proof. The following (stronger claim) also holds: if $a \xrightarrow{i} b$ and $a \xrightarrow{j} c$ then there exists d with $b \xrightarrow{i} d$ and $c \xrightarrow{j} d$. The claim can be shown by induction on the pair (i, j) ordered lexicographically and using the diamond property of $\to$ and the following illustration.

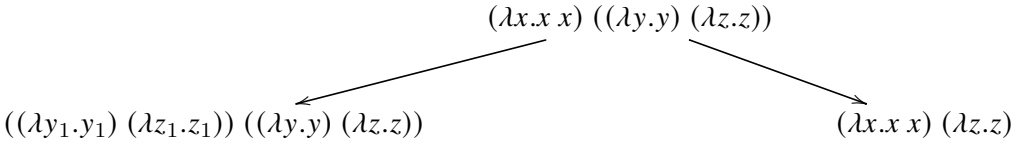


The inner square diagrams follow from the diamond property of $\to$. Since they are squares, there is a one-to-one correspondence w.r.t. the numbers of steps. □

We define some notation for different closures of β-reduction:

Definition 3.3.4. *With $s \xleftrightarrow{C,\beta} t$ we denote the symmetric closure of $\xrightarrow{C,\beta}$ (i.e. $s \xleftrightarrow{C,\beta} t$ iff $s \xrightarrow{C,\beta} t$ or $t \xrightarrow{C,\beta} s$).*
With $s \xrightarrow{C,\beta,*} t$ we denote the reflexive-transitive closure of $\xrightarrow{C,\beta}$ With $s \xleftrightarrow{C,\beta,*} t$ we denote the reflexive-transitive closure of $\xleftrightarrow{C,\beta}$.

Remark 3.3.5. *Note that $\xrightarrow{C,\beta}$ does not have the diamond property:*



The right expression can be reduced to $(\lambda z_1.z_1)\ (\lambda z.z)$ However, the left expression $((\lambda y_1.y_1)\ (\lambda z_1.z_1))\ ((\lambda y.y)\ (\lambda z.z))$ can be reduced to $(\lambda z_1.z_1)\ ((\lambda y.y)\ (\lambda z.z))$ or $((\lambda y_1.y_1)\ (\lambda z_1.z_1))\ (\lambda z.z)$ which are different.

Thus, for proving confluence of $\xrightarrow{\beta}$, we have to find another proof. The idea is to use another reduction relation, called $\to_1$, which can do a bit more than $\xrightarrow{C,\beta}$ or nothing, but not more than $\xrightarrow{C,\beta,*}$, in particular $\xrightarrow{C,\beta,0\vee1} \subseteq \to_1 \subseteq \xrightarrow{C,\beta,*}$ and $\xrightarrow{*}_1 = \xrightarrow{C,\beta,*}$ will hold. Proving that $\xrightarrow{*}_1$ has the diamond property, will then imply that $\xrightarrow{C,\beta,*}$ has the diamond property. The idea of relation $\to_1$ is to reduce an arbitrary number of *parallel* β-redexes in one step.

Definition 3.3.6 (Parallel Reduction $\to_1$)**.** *The relation* $\to_1 \subseteq (\mathbf{Expr} \times \mathbf{Expr})$ *is inductively defined by:*

1. $s \to_1 s$ *for all expressions* s.
2. *if* $s_1 \to_1 s_2$ *and* $t_1 \to_1 t_2$*, then* $(s_1\ t_1) \to_1 (s_2\ t_2)$.
3. *if* $s_1 \to_1 s_2$ *and* $t_1 \to_1 t_2$*, then* $((\lambda x.s_1)\ t_1) \to_1 s_2[t_2/x]$.
4. *if* $s \to_1 t$*, then* $\lambda x.s \to_1 \lambda x.t$.

Lemma 3.3.7. $\xrightarrow{C,\beta} \subseteq \to_1$

Proof. Let $C[(\lambda x.s)\ t] \xrightarrow{C,\beta} C[s[t/x]]$. We show $C[(\lambda x.s)\ t] \to_1 C[s[t/x]]$ by structural induction on C.

If C is the empty context, then $(\lambda x.s)\ t \to_1 s[t/x]$ by Definition 3.3.6, Item 3, since $s \to_1 s$ and $t \to_1 t$ by Definition 3.3.6, Item 1.

For the induction step, we consider several cases where C is not the empty context, and assume as induction hypothesis that $C'[(\lambda x.s)\ t] \to_1 C'[s[t/x]]$ where C' is a proper subcontext of context C.

- If $C = (C'\ r)$, then $r \to_1 r$ by Definition 3.3.6, Item 1, $C'[(\lambda x.s)\ t] \to_1 C'[s[t/x]]$ by the induction hypothesis and thus

  $$C[(\lambda x.s)\ t] = (C'[(\lambda x.s)\ t]\ r) \to_1 (C'[s[t/x]]\ r) = C[s[t/x]]$$

  by Definition 3.3.6, Item 2.
- The case $C = (r\ C')$ is completely analogous to the previous one.
- If $C = \lambda y.C'$, then $C'[(\lambda x.s)\ t] \to_1 C'[s[t/x]]$ by the induction hypothesis and thus

  $$C[(\lambda x.s)\ t] = \lambda y.C'[(\lambda x.s)\ t] \to_1 \lambda y.C'[s[t/x]] = C[s[t/x]]$$

  by Definition 3.3.6, Item 4. □

Example 3.3.8. *We consider the expressions from Remark 3.3.5.*

$$(\lambda x.x\ x)\ ((\lambda y.y)\ (\lambda z.z)) \to_1 ((\lambda y_1.y_1)\ (\lambda z_1.z_1))\ ((\lambda y.y)\ (\lambda z.z))$$

and

$$(\lambda x.x\ x)\ ((\lambda y.y)\ (\lambda z.z)) \to_1 (\lambda x.x\ x)\ (\lambda z.z),$$

since both reductions are also $\xrightarrow{C,\beta}$*-reductions.*

For $(\lambda x.x\ x)\ (\lambda z.z)$ *,we also have* $(\lambda x.x\ x)\ (\lambda z.z) \to_1 (\lambda z_1.z_1)\ (\lambda z.z)$

For $((\lambda y_1.y_1)\ (\lambda z_1.z_1))\ ((\lambda y.y)\ (\lambda z.z))$*, we can reduce the parallel* β*-redexes in on step, which also results in* $(\lambda z_1.z_1)\ (\lambda z.z)$

Lemma 3.3.9. *If* $t \to_1 u$ *then for all* s*:* $s[t/x] \to_1 s[u/x]$.

Proof. By structural induction on s.

- If $s = x$ then $s[t/x] = t \to_1 u = s[u/x]$.
- If $s = y \neq x$, then $s[t/x] = y = s[u/x]$ and the claim holds, since $r \to_1 r$ for all r.
- If $s = (s_1\ s_2)$ then by the induction hypothesis, we have $s_i[t/x] \to_1 s_i[u/x]$ for $i = 1, 2$. Then $s[t/x] = (s_1\ s_2)[t/x] = (s_1[t/x]\ s_2[t/x]) \to_1 (s_1[u/x]\ s_2[u/x]) = (s_1\ s_2)[u/x] = s[u/x]$.
- If $s = \lambda x.s'$, then $s[t/x] = \lambda x.s' = s[u/x]$ the claim holds, since $r \to_1 r$ for all r.
- If $s = \lambda y.s'$, with $y \neq x$, then $s[t/x] = \lambda y.s'[t/x] \to_1 \lambda y.s'[u/x]$ by the induction hypothesis and since $t \to_1 u$. Since $s[u/x] = \lambda y.s'[u/x]$, the claim holds. □

Lemma 3.3.10. *If $s \to_1 r$ and $t \to_1 u$ then $s[t/x] \to_1 r[u/x]$.*

Proof. This can be shown by induction on $s \to_1 r$. For the base case $s = r$ the claim follows by Lemma 3.3.9. For the induction step, we consider several cases:

- If $s = (s_1\ s_2) \to_1 (r_1\ r_2) = r$ with $s_i \to_1 r_i$ for $i = 1, 2$, the induction hypothesis shows $s_i[t/x] \to_1 r_i[u/x]$ for $i = 1, 2$ and thus $(s_1[t/x]\ s_2[t/x]) \to_1 (r_1[u/x]\ r_2[u/x])$. Since

  $$s[t/x] = (s_1\ s_2)[t/x] = (s_1[t/x]\ s_2[t/x]) \to_1 (r_1[u/x]\ r_2[u/x]) = (r_1\ r_2)[u/x] = r[u/x],$$

  the claim holds.
- If $s = (\lambda y.s_1)\ s_2 \to_1 r_1[r_2/y]$ with $s_1 \to_1 r_1$ and $s_2 \to_1 r_2$ and $y \neq x$, the induction hypothesis shows $s_1[t/x] \to_1 r_1[u/x]$ and $s_2[t/x] \to_1 r_2[u/x]$. Then $(\lambda y.s_1[t/x])\ s_2[t/x] \to_1 r_1[u/x][r_2[u/x]/y]$. Since

  $$\begin{aligned} s[t/x] = ((\lambda y.s_1)\ s_2)[t/x] &= (\lambda y.s_1[t/x])\ s_2[t/x] \\ &\to_1 r_1[u/x][r_2[u/x]/y] = r_1[r_2/y][u/x] = r[u/x], \end{aligned}$$

  the claim holds.
- If $s = (\lambda x.s_1)\ s_2 \to_1 r_1[r_2/x]$ with $s_1 \to_1 r_1$ and $s_2 \to_1 r_2$, then $s[t/x] = (\lambda x.s_1)\ s_2[t/x]$ and $r[u/x] = r_1[r_2[u/x]/x]$. The induction hypothesis shows that $s_2[t/x] \to_1 r_2[u/x]$ and thus $s[t/x] = (\lambda x.s_1)\ s_2[t/x] \to_1 r_1[r_2[u/x]/x] = r[u/x]$.
- If $s = \lambda x.s' \to_1 \lambda x.r' = r$ and $s' \to_1 r'$. Then $s[t/x] = s$ and $r[u/x] = r$ and the claim holds.
- If $s = \lambda y.s' \to_1 \lambda y.r' = r$ and $s' \to_1 r'$ where $x \neq y$, then the induction hypothesis shows $s'[t/x] \to_1 r'[u/x]$ and thus $s[t/x] = \lambda y.s'[t/x] \to_1 \lambda y.r'[u/x] = r[u/x]$ □

Lemma 3.3.11. *Relation $\to_1$ has the diamond property.*

Proof. We show that whenever $s \to_1 t$ then for all r with $s \to_1 r$ there exists r' with $t \to_1 r'$ and $r \to r'$. We use induction on the definition of $\to_1$ in $s \to_1 t$.

Base case: $t = s$, i.e. $s \to_1 s$. Then choose $r' = r$ and the claim holds.

For the induction step, we consider the other cases of the definition of $\to_1$:

- If $s = (s_1\ s_2), t = (t_1\ t_2), s_1 \to_1 t_1$ and $s_2 \to_1 t_2$, then for $s \to_1 r$ there are the following cases:
  - $s = (s_1\ s_2) \to_1 (r_1\ r_2)$ with $s_1 \to_1 r_1$ and $s_2 \to_1 r_2$. The induction hypothesis applied twice (to s_1 and s_2) gives us r_1', r_2' with $t_i \to r_i', r_i \to r_i'$ for $i = 1, 2$. Let $r' = (r_1'\ r_2')$. Then the definition of $\to_1$ shows that $t \to_1 r'$ and $r \to_1 r$. Thus we are finished.
  - $s_1 = \lambda x.s_1'$ and $s_1' \to_1 r_1$, $s_2 \to_1 r_2$, and $r = s_1'[r_2/x]$. Then $t_1 = \lambda x.t_1'$ where $s_1' \to_1 t_1'$ (other cases are not possible).

    Applying the induction hypothesis to $s_1' \to_1 t_1'$ and $s_1' \to_1 r_1$ shows that there is r_1' with $t_1' \to_1 r_1'$ and $r_1 \to_1 r_1'$. Applying the induction hypothesis to $s_2 \to t_2$ and $s_2 \to r_2$ show that there is r_2' with $t_1 \to_1 r_2'$ and $r_2 \to r_2'$.

    This shows that $t = (t_1\ t_2) = ((\lambda x.t_1')\ t_2) \to_1 r_1'[r_2'/x]$ and also $r = s_1'[r_2/x] \to_1 r_1'[r_2'/x]$ (where we apply Lemma 3.3.10).
- If $s = ((\lambda x.s_1)\ s_2)$ and $t = t_1[t_2/x]$ where $s_1 \to_1 t_1$ and $s_2 \to_1 t_2$. Then for $s \to_1 r$ there are again two cases:
  - $s = ((\lambda x.s_1)\ s_2) \to_1 ((\lambda x.r_1)\ r_2)$ with $s_1 \to_1 r_1$ and $s_2 \to_1 r_2$. Applying the induction hypothesis to $s_1 \to_1 t_1$ and $s_1 \to_1 r_1$ and also to $s_2 \to_1 t_2$ and $s_2 \to_1 r_2$ shows that there exists r_1' and r_2' such that: $t_i \to_1 r_i', r_i \to_1 r_i'$ for $i = 1, 2$. This shows that $t_1[t_2/x] \to_1 r_1'[r_2'/x]$ and $((\lambda x.r_1)\ r_2) \to_1 r_1'[r_2'/x]$ (using Lemma 3.3.10). Thus the diamond property holds.
  - If $s = ((\lambda x.s_1)\ s_2)$ and $r = r_1[r_2/x]$ where $s_1 \to_1 r_1$ and $s_2 \to_1 r_2$. Applying the induction hypothesis to $s_1 \to_1 t_1$ and $s_1 \to_1 r_1$ and also to $s_2 \to_1 t_2$ and $s_2 \to_1 r_2$ shows that there exists r_1' and r_2' such that: $t_i \to_1 r_i'$, $r_i \to_1 r_i'$ for $i = 1, 2$. This shows that $t_1[t_2/x] \to_1 r_1'[r_2'/x]$ and $r_1[r_2/x] \to_1 r_1'[r_2'/x]$ (using Lemma 3.3.10). Thus the diamond property holds.
- If $s = \lambda x.s_1, t = \lambda x.t_1$, and $s_1 \to_1 t_1$, then for $s \to_1 r$ only $r = \lambda x.r_1$ with $s_1 \to_1 r_1$ can hold. Applying the induction hypothesis to $s_1 \to_1 t_1$ and $s_1 \to_1 r_1$ shows that there exists r_1' with $t_1 \to_1 r_1'$ and $r_1 \to r_1'$. Definition of $\to_1$ thus shows $\lambda x.t_1 \to \lambda x.r_1'$ and $\lambda x.r_1 \to \lambda x.r_1'$. Thus the diamond property holds. □

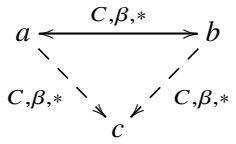
Lemma 3.3.12. $\xrightarrow{C,\beta,*} = \overset{*}{\to}_1$

Proof sketch. Since $\xrightarrow{C,\beta} \subseteq \to_1$ (Lemma 3.3.7), $\xrightarrow{C,\beta,*} \subseteq \overset{*}{\to}_1$ also holds.

The inclusion $\to_1 \subseteq \xrightarrow{C,\beta,*}$ can be proved by inspecting the different cases of the inductive definition of $\to_1$. Finally, $\overset{*}{\to}_1 \subseteq \xrightarrow{C,\beta,*}$ holds, since $\overset{*}{\to}_1 \subseteq \left(\xrightarrow{C,\beta,*}\right)^*$ and $\xrightarrow{C,\beta,*}{}^* = \xrightarrow{C,\beta,*}$. □

We are now able to show that $\xrightarrow{C,\beta}$ is confluent, which follows from the following theorem:

Theorem 3.3.13 (Church-Rosser-Theorem)**.** *For the lambda calculus the following holds: If* $a \overset{C,\beta,*}{\longleftrightarrow} b$*, then there exists c, such that* $a \overset{C,\beta,*}{\longrightarrow} c$ *and* $b \overset{C,\beta,*}{\longrightarrow} c$



(Note that this result is up to α*-equivalence.)*

Proof. Applying Lemma 3.3.3 for $\to_1$ (using Lemma 3.3.11) shows that $\to_1$ is confluent and that $\overset{*}{\to}_1$ has the diamond property. With the equation of Lemma 3.3.12, we have that $\overset{C,\beta,*}{\longrightarrow}$ has the diamond property and thus $\overset{C,\beta}{\longrightarrow}$ is confluent. Finally, Lemma 3.3.2 then shows the claim. □

One nice property of confluence is that normal forms are unique, i.e. if we β-reduce a λ-expression into an expression that has no more β-redexes, then we always get the same expression (up to α-renaming), independently from the order and positions where the reductions were applied.

3.4. Call-by-Name Evaluation

The *call-by-name evaluation* always reduces the leftmost-outermost β-redex. The idea is to evaluate an application of an abstraction to an argument *without* evaluating the argument, but immediately passing the argument to the body of the abstraction.

Formally, we define call-by-name reduction using reduction contexts:

Definition 3.4.1. *Reduction contexts R are built by the following grammar with start symbol* **RCtxt** *(where* **Expr** *are generated as defined in Definition 3.1.1):*

$$\textbf{RCtxt} ::= [\cdot] \mid (\textbf{RCtxt}\ \textbf{Expr})$$

If $r_1 \overset{\beta}{\to} r_2$ *and R is a reduction context, then* $R[r_1] \overset{name}{\longrightarrow} R[r_2]$ *is a* call-by-name reduction step.

Exercise 3.4.2. *Let* $s = (\lambda w.w)\ (\lambda x.x)\ ((\lambda y.((\lambda u.y)\ y))\ (\lambda z.z))$*. Write down all reduction contexts R and expressions t such that* $R[t] = s$*. Perform a call-by-name reduction step for expressions s*

Remark 3.4.3. *When a closed expressions is reduced using call-by-name reduction steps, then in principle, one can omit the* α*-renaming, since capturing free variables is impossible for this case. But: if* β*-steps are performed, which are not call-by-name reduction steps, then the* α*-renaming is needed to avoid unwanted (and wrong) captures of free variables.*

An alternative method for performing call-by-name reduction is the following (intuitive) algorithm using labels on the expressions.

Let s be an expression. Label s with a star, i.e. $s^\star$.

Now, perform the following shifting of the label as long as possible:

$$(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$$

The result is of the form $(s_1^\star\ s_2\ \ldots\ s_n)$, where s_1 is not an application. There are the following cases:

- s_1 is an abstraction $\lambda x.s_1'$: If $n \geq 2$, then reduce s as follows:

$$(\lambda x.s_1')\ s_2\ \ldots s_n \xrightarrow{name} (s_1'[s_2/x]\ \ldots s_n).$$

  If $n = 1$, then no call-by-name reduction is possible (since the whole expression is an abstraction).
- s_1 is a variable. Then no call-by-name reduction is applicable, but a free variable has been detected (for closed expressions, this case does not occur).

As an example, we consider $((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))$. The labeling algorithm shifts the label as follows:

$$((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))^\star \Rightarrow ((\lambda x.\lambda y.x)^\star((\lambda w.w)(\lambda z.z)))$$

Now a call-by-name reduction is possible, since the subexpression labeled with $\star$ is an abstraction which is applied to an argument, i.e. the call-by-name reduction is:

$$((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z))) \xrightarrow{name} \lambda y.((\lambda w.w)(\lambda z.z))$$

Since, this first step result in an abstraction, no further call-by-name reduction step is applicable and the call-by-name evaluation stops.

Exercise 3.4.4. *Evaluate the following expressions with call-by-name evaluation (i.e. perform call-by-name reduction steps as long as possible)*

- $(\lambda f.(\lambda x.f\ (x\ x)))\ (\lambda.f.(\lambda x.f\ (x\ x)))(\lambda w.\lambda z.w)$
- $(\lambda f.(\lambda x.f\ (x\ x)))\ (\lambda.f.(\lambda x.f\ (x\ x)))((\lambda z.z)\ (\lambda w.w))$

Call-by-name reduction is deterministic, i.e. for expression s, there is at most one expression such that $s \xrightarrow{name} t$. There are expressions where no reduction is possible, which is the case if evaluation detects a free variable in function position (for example $(x\ (\lambda y.y))$, in general, the expression is of the form $R[x]$ where R is a reduction context), or if the expression is an abstraction. Abstractions are also called *FWHNFs* (functional weak head normal forms). If we reach an FWHNF using $\xrightarrow{name}$-reduction, the call-by-name evaluation successfully stops. Expressions which evaluate to an abstraction are called *converging* (or alternatively, are *terminating*).

Let $\xrightarrow{name,+}$ be the transitive closure of $\xrightarrow{name}$ and $\xrightarrow{name,*}$ be the reflexive-transitive closure of $\xrightarrow{name}$.

Definition 3.4.5. *An expression s (call-by-name)* converges *iff there is a sequence of call-by-name reduction steps starting with s that ends in an abstraction. In this case, we write $s{\downarrow}$. I.e.*

$$s{\downarrow} \text{ iff } \exists \text{abstraction } v : s \xrightarrow{name,*} v.$$

If s does not converge, we write $s{\Uparrow}$ and say s diverges.

In the remainder of this section, we will show that call-by-name evaluation is an optimal strategy with respect to termination. For the proof, we again use the parallel reduction $\to_1$, where we use $\to_1$ also on contexts: contexts are treated like an expression where the hole $[\cdot]$ is treated like a new constant c, and then usual $\to_1$-reduction on expressions is performed.

Lemma 3.4.6. *For $\to_1$-reduction on contexts the following holds:*

1. *If $R \to_1 R'$ for a reduction context R, then R' is also a reduction context.*
2. *If $C \to_1 C'$, $s \to_1 s'$, where C is a context, then $C[s] \to_1 C[s']$*

Proof. This can be shown by structural induction on R or C, respectively. □

We define the relation $\xrightarrow{name}_1$ where the idea is, that it is a $\to_1$-step that includes a $\xrightarrow{name}$-reduction. In contrast, the relation $\xrightarrow{int}_1$ represents a $\to_1$-step that does not contain a $\xrightarrow{name}$-reduction.

Definition 3.4.7. *If $R \to_1 R'$, $s \to_1 s'$, $t \to_1 t'$, and R is a reduction context, then $R[(\lambda x.s)\ t] \xrightarrow{name}_1 R'[s'[t'/x]]$.*
Let $\xrightarrow{int}_1 := \to_1 \setminus \xrightarrow{name}_1$ be the internal $\xrightarrow{1}$-reduction.

As an example, consider the expression $s = (\lambda x.(\lambda u.(u\ u)\ (\lambda b.b)))\ ((\lambda w.w)\ (\lambda a.a))$. There are the following expressions t_i with $s \to_1 t_i$:

1. $t_1 = s$
2. $t_2 = (\lambda u.(u\ u)\ (\lambda b.b))$
3. $t_3 = (\lambda x.((\lambda b.b)\ (\lambda b.b)))\ ((\lambda w.w)\ (\lambda a.a))$
4. $t_4 = (\lambda x.(\lambda u.(u\ u)\ (\lambda b.b)))\ (\lambda a.a))$
5. $t_5 = ((\lambda b.b)\ (\lambda b.b))$
6. $t_6 = (\lambda x.((\lambda b.b)\ (\lambda b.b)))\ (\lambda a.a)$

From the six step $s \to_1 t_i$, the steps $s \to_1 t_2$ and $s \to_1 t_5$ are also $\xrightarrow{name}_1$-steps, all other steps are $\xrightarrow{int}_1$-steps.

An easy consequence of the definitions is:

Lemma 3.4.8. *The following chain of inclusions holds: $\xrightarrow{name} \subset \xrightarrow{name}_1 \subset \to_1$*

Before proving the next lemma, we introduce a measure, which counts the number of β-redexes that are contracted by a single $\to_1$-step. Note that we cannot define the measure on the left expression of the reduction step, we need to consider the full step, since $\to_1$ allows to reduce any subset of all parallel β-redexes.

Definition 3.4.9. *Define the measure $\phi : \to_1 \to \mathbb{N}_0$ inductively as*

$$\begin{array}{ll}
\phi(x \to_1 x) & = 0, \textit{if } x \textit{ is a variable} \\
\phi(\lambda x.s \to_1 \lambda x.s') & = \phi(s \to s') \\
\phi(((\lambda x.s)\ t) \to_1 s'[t'/x]) & = 1 + \phi(s \to_1 s') + k \cdot \phi(t \to_1 t'), \textit{where } k \textit{ is the number of} \\
 & \qquad\qquad \textit{free occurrences of } x \textit{ in } s \\
\phi((s\ t) \to_1 (s'\ t')) & = \phi(s \to_1 s') + \phi(t \to_1 t')
\end{array}$$

We observe, that the measure ϕ is defined for every $\to_1$-step and that it is well-founded, since it cannot be smaller than 0.

The following lemma shows that a $\to_1$ step can be split into a sequence of call-by-name steps followed by an internal $\to_1$ -step.

Lemma 3.4.10. *If $s \to_1 t$, then $s \xrightarrow{name,*} s' \xrightarrow{int}_1 t$.*

Proof. Let $s \to_1 t$. If the reduction is internal, i.e. $s \xrightarrow{int}_1 t$, then the claim holds.

Otherwise, the reduction is a $\xrightarrow{name}_1$-reduction. Then there exist reduction contexts R and R' such that $s = R[(\lambda x.r)\ u]$, $r \to_1 r'$, $u \to_1 u'$, $R \to_1 R'$, and $t = R'[r'[u'/x]]$.

Then $s \xrightarrow{name} R[r[u/x]] \to_1 R'[r'[u'/x]]$ and this process can be iterated (starting with $R[r[u/x]]$ in the next iteration).

Thus if the iteration stops, the demanded reduction sequence is constructed, and the claim of the lemma holds.

However, we have to argue that it is terminating: We use the measure ϕ and show that $\phi(s \to_1 t) > \phi(R[r[u/x]] \to_1 R'[r'[u'/x]])$. Since ϕ is well-founded, the iteration must terminate.

We have

$$\begin{array}{rl}
 & \phi(R[(\lambda x.r)\ u] \to_1 R'[r'[u'/x]]) \\
= & \phi(R \to_1 R') + \phi((\lambda x.r)\ u \to_1 r'[u'/x]) \\
= & \phi(R \to_1 R') + 1 + \phi(r \to_1 r') + k\phi(u \to_1 u')
\end{array}$$

where k is the number of free occurrences of x in r.

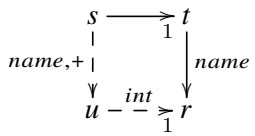
For $R[r[u/x]] \to_1 R'[r'[u'/x]]$, we have

$$\begin{array}{rl}
 & \phi(R[r[u/x]]] \to_1 R'[r'[u'/x]]) \\
= & \phi(R \to_1 R') + \phi(r \to_1 r') + k\phi(u \to_1 u')
\end{array}$$

where still k is the number of free occurrences of x in r.

Thus, the measure is strictly decreased, which shows the claim. □

Lemma 3.4.11. *Let s, t, r be expressions such that $s \to_1 t \xrightarrow{name} r$, then there exists u such that $s \xrightarrow{name,+} u \xrightarrow{int}_1 r$.*



Proof. We apply Lemma 3.4.10 to $s \to_1 t$ and thus the sequence $s \to_1 t \xrightarrow{name} r$ can be rewritten as $s \xrightarrow{name,*} t' \xrightarrow{int}_1 t \xrightarrow{name} r$.

Since $t \xrightarrow{name} r$ we can assume that $t = R[(\lambda x.t_0)\ t_1] \xrightarrow{name} R[t_0[t_1/x]] = r$ for some expressions t_0, t_1 and reduction context R. Since $t' \xrightarrow{int}_1 t''$ is internal, the following must hold $t' = R'[(\lambda x.t_0')\ t_1']$ where $R' \to_1 R$, $t_0' \to_1 t_0$, and $t_1' \to_1 t_1$. But then $t' = R'[(\lambda x.t_0')\ t_1'] \xrightarrow{name} R'[t_0'[t_1'/x]] \to_1 R[t_0[t_1/x]] = r$ holds. Applying Lemma 3.4.10 to $R'[t_0'[t_1'/x]] \to_1 r$ finally shows that $s \xrightarrow{name,*} R'[t_0'[t_1'/x]] \xrightarrow{name,*} R[t_0[t_1/x]] \xrightarrow{int}_1 r$, which shows the claim. □

Lemma 3.4.12. *If $s \to_1 t \xrightarrow{name,*} r$ where r is an FWHNF, then there exists a sequence $s \xrightarrow{name,*} r' \xrightarrow{int}_1 r$ where r' is an FWHNF.*

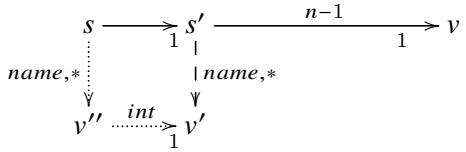
Proof. For $s \to_1 t \xrightarrow{name,j} r$, apply Lemma 3.4.11 j-times to shift the $\to_1$-reduction over each $\xrightarrow{name}$-reduction. □

Call-by-name reduction has the following property:

Theorem 3.4.13 (Call-by-name evaluation is standardising)**.** *Let s be an expression. If s can be transformed into an abstraction using* arbitrary *β-reduction steps (at any position), then $s{\downarrow}$.*

Proof. The given sequence is also a sequence of $\to_1$-reductions, since $\xrightarrow{C,\beta} \subseteq \to_1$. Thus it suffices to show that if $s \xrightarrow{n}_1 v$ where v is an FWHNF, then $s{\downarrow}$. We use induction on n. If $n = 0$, then the claim holds. For the induction step, let $s \to_1 s' \xrightarrow{n-1}_1 v$ where v is an FWHNF. By the induction hypothesis we get that $s'{\downarrow}$, i.e. $s' \xrightarrow{name,*} v'$ where v' is an FWHNF. Now apply Lemma 3.4.12 to $s \to_1 s' \xrightarrow{name,*} v'$ which shows that $s \xrightarrow{name,*} v'' \xrightarrow{int}_1 v'$ where v'' is an FWHNF. Hence we have $s{\downarrow}$.

The induction step can be depicted as follows, where the dashed reductions follow from the induction hypothesis and the dotted reductions follow from Lemma 3.4.12.



□

The theorem shows, that call-by-name evaluation is an optimal strategy w.r.t. termination.

Example 3.4.14. *We show an example how the proof of Theorem 3.4.13 constructs the call-by-name evaluation for the given sequence*

$$\begin{array}{ll} & (\lambda x.x\ \lambda c.((\lambda u.u\ u)\ (\lambda b.b)))\ ((\lambda y.y\ ((\lambda w.w)\ (\lambda z.z)))\ (\lambda a.a)) \\ \to_1 & (\lambda a.a)\ (\lambda z.z)\ (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \to_1 & (\lambda z.z)\ (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \to_1 & \lambda c.((\lambda b.b)\ (\lambda b.b)) \\ \to_1 & \lambda c.(\lambda b.b) \end{array}$$

It replaces the $\to_1 \xrightarrow{name,*}$ steps by $\xrightarrow{name,*}\xrightarrow{int}{}_1$-sequences using Lemma 3.4.12, by always replacing the right-most $\to_1$-step.

1. *Replacing the last $\to_1$-step of the given sequence results in*

$$\begin{array}{ll} & (\lambda x.x\ \lambda c.((\lambda u.u\ u)\ (\lambda b.b)))\ ((\lambda y.y\ ((\lambda w.w)\ (\lambda z.z)))\ (\lambda a.a)) \\ \to_1 & (\lambda a.a)\ (\lambda z.z)\ (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \to_1 & (\lambda z.z)\ (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \to_1 & \lambda c.((\lambda b.b)\ (\lambda b.b)) \\ \xrightarrow{int}_1 & \lambda c.(\lambda b.b) \end{array}$$

   Note that the sequence of call-by-name-steps is empty (since the $\to_1$-step was an internal $\to_1$-step).

2. *Replacing the rightmost $\to_1$-step then results in*

$$\begin{array}{ll} & (\lambda x.x\ \lambda c.((\lambda u.u\ u)\ (\lambda b.b)))\ ((\lambda y.y\ ((\lambda w.w)\ (\lambda z.z)))\ (\lambda a.a)) \\ \to_1 & (\lambda a.a)\ (\lambda z.z)\ (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \to_1 & (\lambda z.z)\ (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \xrightarrow{name} & (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \xrightarrow{int}_1 & \lambda c.((\lambda b.b)\ (\lambda b.b)) \\ \xrightarrow{int}_1 & \lambda c.(\lambda b.b) \end{array}$$

3. *Replacing the rightmost $\to_1$-step then results in*

$$\begin{array}{ll} & (\lambda x.x\ \lambda c.((\lambda u.u\ u)\ (\lambda b.b)))\ ((\lambda y.y\ ((\lambda w.w)\ (\lambda z.z)))\ (\lambda a.a)) \\ \to_1 & (\lambda a.a)\ (\lambda z.z)\ (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \xrightarrow{name} & (\lambda z.z)\ (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \xrightarrow{name} & (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \xrightarrow{int}_1 & (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \xrightarrow{int}_1 & \lambda c.((\lambda b.b)\ (\lambda b.b)) \\ \xrightarrow{int}_1 & \lambda c.(\lambda b.b) \end{array}$$

4. *Replacing the rightmost* $\to_1$*-step then results in*

$$\begin{array}{ll} & (\lambda x.x\ \lambda c.((\lambda u.u\ u)\ (\lambda b.b)))\ ((\lambda y.y\ ((\lambda w.w)\ (\lambda z.z)))\ (\lambda a.a)) \\ \xrightarrow{name} & ((\lambda y.y\ ((\lambda w.w)\ (\lambda z.z)))\ (\lambda a.a))\ \lambda c.((\lambda u.u\ u)\ (\lambda b.b)) \\ \xrightarrow{name} & ((\lambda a.a)\ ((\lambda w.w)\ (\lambda z.z)))\ \lambda c.((\lambda u.u\ u)\ (\lambda b.b)) \\ \xrightarrow{name} & ((\lambda w.w)\ (\lambda z.z))\ \lambda c.((\lambda u.u\ u)\ (\lambda b.b)) \\ \xrightarrow{name} & ((\lambda z.z)\ \lambda c.((\lambda u.u\ u)\ (\lambda b.b)) \\ \xrightarrow{name} & (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \xrightarrow{int}_1 & (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \xrightarrow{int}_1 & (\lambda c.((\lambda u.u\ u)\ (\lambda b.b))) \\ \xrightarrow{int}_1 & \lambda c.((\lambda b.b)\ (\lambda b.b)) \\ \xrightarrow{int}_1 & \lambda c.(\lambda b.b) \end{array}$$

3.5. Call-by-Value Evaluation

An import other evaluation strategy which is used in strict functional programming languages is *call-by-value evaluation* (sometimes called *strict evaluation*). The main difference compared to call-by-name evaluation, is that β-reduction is permitted only if the argument applied to the abstraction is already a value (and thus also an abstraction or a variable). So we define:

Definition 3.5.1. *The (direct)* (β_{value})-reduction *is defined as*

$$(\beta_{value}) \qquad (\lambda x.s)\ v \xrightarrow{\beta_{value}} s[v/x],\ \textit{where } v \textit{ is a variable or an abstraction}$$

The contextual closure of β_{value}*-reduction is* $\xrightarrow{C,\beta_{value}}$ *defined as*

$$C[s] \xrightarrow{C,\beta_{value}} C[t] \textit{ iff } C \textit{ is a context and } s \xrightarrow{\beta_{value}} t.$$

Clearly, $\xrightarrow{\beta_{value}} \subset \xrightarrow{\beta}$.

To proceed e.g. with the evaluation of the expression $(\lambda x.x)\ s$ where $s = ((\lambda y.(y\ y))\ (\lambda z.z))$, call-by-value evaluation is not allowed to reduce $(\lambda x.x)\ s$, since s is not a value. The strategy of call-by-value evaluation can thus be described by "evaluate parameters before calling the function" (in contrast, for call-by-name evaluation we call the function without evaluating the parameters).

To defined this strategy we again use contexts:

Definition 3.5.2. Call-by-value reduction context*s E are built by the following grammar with start symbol* **ECtxt** *(where* **Expr** *are generated as defined in Definition 3.1.1)*

$$\textbf{ECtxt} ::= [\cdot] \mid (\textbf{ECtxt}\ \textbf{Expr}) \mid ((\lambda V.\textbf{Expr})\ \textbf{ECtxt})$$

If $r_1 \xrightarrow{\beta_{value}} r_2$ *and* E *is a call-by-value reduction context, then*

$$E[r_1] \xrightarrow{value} E[r_2]$$

is a call-by-value reduction.

As alternative definition of call-by-value reduction, we can again use a labeling algorithm. If s is an expression, then begin with $s^\star$ and apply the following label-shifting rules exhaustively:

$(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$
$(v^\star\ s) \Rightarrow (v\ s^\star)$ if v is an abstraction and s is not an abstraction or a variable

If in the result a variable is labeled with $\star$, then no reduction is applicable, since a free variable was found in reduction position (i.e. the expression s is of the form $E[x]$). If s is labeled with $\star$, then s is a variable or an abstraction, and no reduction is possible (since also for call-by-value reduction, abstractions are the successful values, as we will see below). Otherwise, an abstraction is labeled with $\star$ and the direct superterm is an application where the argument is an abstraction or a variable (i.e. $s = E[((\lambda x.s')\ v)]$ and labeling stopped with $E[((\lambda x.s')^\star\ v)]$). Call-by-value evaluation reduces this application.

Let us consider the expression $((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))$. Searching the redex using the labeling algorithm performs the following shiftings:

$$\begin{aligned}
&((\lambda x.\lambda y.x)\ ((\lambda w.w)\ (\lambda z.z)))^\star \\
\Rightarrow&((\lambda x.\lambda y.x)^\star\ ((\lambda w.w)\ (\lambda z.z))) \\
\Rightarrow&((\lambda x.\lambda y.x)\ ((\lambda w.w)\ (\lambda z.z))^\star) \\
\Rightarrow&((\lambda x.\lambda y.x)\ ((\lambda w.w)^\star\ (\lambda z.z)))
\end{aligned}$$

The first call-by-vale reduction is thus:

$$((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z))) \xrightarrow{value} (\lambda x.\lambda y.x)(\lambda z.z)$$

Now the argument of the topmost application is a value and thus the whole expression is the redex of the subsequent call-by-value β-reduction, resulting in

$$(\lambda x.\lambda y.x)(\lambda z.z) \xrightarrow{value} \lambda y.\lambda z.z$$

Now an abstraction is obtained and call-by-value evaluation successfully stops.

Using induction on the term structure, it is easy to verify that call-by-value reduction is deterministic and hence unique. Values are FWHNFs (i.e. abstractions). Convergence is then defined as follows:

Definition 3.5.3. *An expression* s converges for call-by-value evaluation *iff there is sequence of call-by-value reduction, that transforms* s *into an abstraction. In this case we write* $s{\downarrow_{value}}$. *I.e.,*

$s{\downarrow_{value}}$ *iff* $\exists$*abstraction* $v : s \xrightarrow{value,*} v$. *If* s *does not call-by-value converge, we write* $s{\Uparrow_{value}}$ *and say* s diverges for call-by-value evaluation.

From Theorem 3.4.13 immediately follows: $s{\downarrow_{value}} \implies s{\downarrow}$. The reverse implication does not hold, since there exist expressions that diverge for call-by-value evaluation, but converge using call-by-name evaluation:

Example 3.5.4. *Consider the expression* $\Omega := (\lambda x.x\ x)\ (\lambda x.x\ x)$. *It is easy to verify that* $\Omega \xrightarrow{name} \Omega$ *and also* $\Omega \xrightarrow{value} \Omega$. *Hence we have* $\Omega{\Uparrow}$ *and* $\Omega{\Uparrow_{value}}$. *Now consider the expression* $t := ((\lambda x.(\lambda y.y))\ \Omega)$. *Then* $t \xrightarrow{name} \lambda y.y$, *i.e.* $t{\downarrow}$. *But call-by-value evaluation will first evaluate the argument* Ω *which does not terminate (i.e.* $t \xrightarrow{value} t \xrightarrow{value} \cdot$*). Thus* $t{\Uparrow_{value}}$.

The order of evaluation is predictable for call-by-value evaluation. For instance, when applying function f to closed expressions s_1, s_2, s_3, then first s_1, s_2, s_3 are evaluated sequentially and then the application of f to the values is evaluated. For call-by-name evaluation the exact order, when s_1, s_2 and s_3 are evaluated (if at all!) depends on the definition of f. Since the order is predictable for call-by-value evaluation, side-effects like printing on a screen etc. is often done directly in strict functional languages, while in non-strict languages like Haskell more effort is necessary to control (and this execute) side-effects. Some prominent examples of strict functional programming languages are ML (with dialects SML, OCaml), Scheme and Microsofts F#.

3.6. Call-by-Need Evaluation

Call-by-need evaluation can be seen as an optimization of call-by-name evaluation. To keep things simple, we introduce a variant of the call-by-need evaluation, which uses a new syntactic construct (i.e. `let`-expressions). However, even with this construct, evaluation is quite complicated.

Definition 3.6.1. *The syntax of expressions with* `let` *is given by the following grammar with start non-terminal* **Expr***:*

$$\mathbf{Expr} ::= V \mid \lambda V.\mathbf{Expr} \mid (\mathbf{Expr}\ \mathbf{Expr}) \mid \texttt{let}\ V = \mathbf{Expr}\ \texttt{in}\ \mathbf{Expr}$$

The scope of x in `let` $x = s$ `in` t is t, i.e. our `let`-expressions are not recursive. For the distinct variable convention, we thus assume that $x \notin FV(s)$.

Call-by-need reduction contexts R_{need} are generated by the following grammar with start symbol $\mathbf{R}_{need}$. It uses two further context classes, built by the non-terminals **A** and **LR**. The non-terminal **Expr** represents expressions built according to Definition 3.6.1.

$$\begin{array}{rcl} \mathbf{R}_{need} & ::= & \mathbf{LR}[\mathbf{A}] \mid \mathbf{LR}[\texttt{let}\ x = \mathbf{A}\ \texttt{in}\ \mathbf{R}_{need}[x]] \\ \mathbf{A} & ::= & [\cdot] \mid (\mathbf{A}\ \mathbf{Expr}) \\ \mathbf{LR} & ::= & [\cdot] \mid \texttt{let}\ V = \mathbf{Expr}\ \texttt{in}\ \mathbf{LR} \end{array}$$

The **A**-contexts are like call-by-name reduction contexts (the path to the context whole always chooses the left argument of an application). The **LR**-contexts have their hole in the `in`-part of the `let`-expressions[1]. Reduction contexts first exhaustively walk through the `in`-expressions of (perhaps nested) `let`-expressions. If needed, `let`-bindings are visited, if the value of a binding $x = t$ is needed.

Example 3.6.2. *As an example consider the expression* $s := \texttt{let}\ x = ((\lambda z.z)\ (\lambda u.u))\ \texttt{in}\ \texttt{let}\ y = \lambda\, w.w\ \texttt{in}\ (x\ w)$*: Then there are the following pairs of a reductions contexts R and expressions t, such that* $R[s] = t$*:*

- $R = [\cdot]$ *and* $t = s$
- $R = \texttt{let}\ x = ((\lambda z.z)\ (\lambda u.u))\ \texttt{in}\ [\cdot]$ *and* $t = \texttt{let}\ y = \lambda\, w.w\ \texttt{in}\ (x\ w)$
- $R = \texttt{let}\ x = ((\lambda z.z)\ (\lambda u.u))\ \texttt{in}\ \texttt{let}\ y = \lambda\, w.w\ \texttt{in}\ [\cdot]$ *and* $t = (x\ w)$
- $R = \texttt{let}\ x = ((\lambda z.z)\ (\lambda u.u))\ \texttt{in}\ \texttt{let}\ y = \lambda\, w.w\ \texttt{in}\ ([\cdot]\ w)$ *and* $t = x$
- $R = \texttt{let}\ x = [\cdot]\ \texttt{in}\ \texttt{let}\ y = \lambda\, w.w\ \texttt{in}\ (x\ w)$ *and* $t = ((\lambda z.z)\ (\lambda u.u))$
- $R = \texttt{let}\ x = ([\cdot]\ (\lambda u.u))\ \texttt{in}\ \texttt{let}\ y = \lambda\, w.w\ \texttt{in}\ (x\ w)$ *and* $t = \lambda z.z$

Instead of (β)-reduction, the following reduction rules are used. Each of these steps is a *call-by-need reduction step*, denoted with $\xrightarrow{need}$.

$$
\begin{array}{ll}
(lbeta) & R_{need}[(\lambda x.s)\ t] \xrightarrow{need} R_{need}[\texttt{let}\ x = t\ \texttt{in}\ s] \\
(cp) & LR[\texttt{let}\ x = \lambda y.s\ \texttt{in}\ R_{need}[x]] \xrightarrow{need} LR[\texttt{let}\ x = \lambda y.s\ \texttt{in}\ R_{need}[\lambda y.s]] \\
(llet) & LR[\texttt{let}\ x = (\texttt{let}\ y = s\ \texttt{in}\ t)\ \texttt{in}\ R_{need}[x]] \\
 & \quad \xrightarrow{need} LR[\texttt{let}\ y = s\ \texttt{in}\ (\texttt{let}\ x = t\ \texttt{in}\ R_{need}[x])] \\
(lapp) & R_{need}[(\texttt{let}\ x = s\ \texttt{in}\ t)\ r] \xrightarrow{need} R_{need}[\texttt{let}\ x = s\ \texttt{in}\ (t\ r)]
\end{array}
$$

Here R_{need} is an $\mathbf{R}_{need}$-context and LR is an **LR**-context.

We explain the rules: instead of a substituting (β)-reduction, the $(lbeta)$-reduction is used, which delays substitution: it shares the argument by a new binding. The (cp)-reduction then copies such a binding, if it is needed by the evaluation, but first the right hand side must be evaluated to a value (i.e. an abstraction). This avoids "work duplication". Rules $(llet)$ und $(lapp)$ are used to reorder `let`-nestings.

Remark 3.6.3. *The main difference between call-by-name and call-by-need evaluation is the following idea: For* $R[(\lambda x.s)\ t]$*, call-by-name evaluation substitutes all free occurrences of x in s by t. If an occurrence of x is required for the result of evaluating s, then t is evaluated for every such occurrence. This duplicates work, for instance in* $(\lambda x.((x\ x)\ x))\ t$ *where t evaluates to the identity* $\lambda z.z$*, the work for evaluating t is performed three times after substituting x with t.*

Call-by-need evaluation tries to avoid this kind of work duplication. Instead it delays the evaluation of t and if t is evaluated (since it is required) its result is shared such that evaluation is not repeated. Sharing is implemented by `let`*-bindings.*

[1] The name **LR** stands for "let right"

A labeling algorithm to search the next redex of call-by-need evaluation is the following. For expression s, start with $s^\star$. The algorithm uses further labels: $^\diamond$ and $^\circledcirc$, and the notation $\star \vee \diamond$ means that the label can be * or $^\diamond$. The rules of the label shifting are as defined below, They are applied exhaustively, where in case that rule (2) and rule (3) are applicable always rule (2) is applied.

$$
\begin{array}{ll}
(1) & (\texttt{let}\ x = s\ \texttt{in}\ t)^\star \Rightarrow (\texttt{let}\ x = s\ \texttt{in}\ t^\star) \\
(2) & (\texttt{let}\ x = C_1[y^\diamond]\ \texttt{in}\ C_2[x^\circledcirc]) \Rightarrow (\texttt{let}\ x = C_1[y^\diamond]\ \texttt{in}\ C_2[x]) \\
(3) & (\texttt{let}\ x = s\ \texttt{in}\ C[x^{\star\vee\diamond}]) \Rightarrow (\texttt{let}\ x = s^\diamond\ \texttt{in}\ C[x^\circledcirc]) \\
(4) & (s\ t)^{\star\vee\diamond} \Rightarrow (s^\diamond\ t)
\end{array}
$$

We explain the rules: rule (1) visits the `in`-expression of `let`-expressions. The label $\star$ is flipped to $\diamond$ after the shifting visits a function position of an application or `let`-bindings. This prevents from visiting other `let`-expressions using rule (1). Rule (3) introduces the label $\circledcirc$ to label the copy target of a potential (cp)-reduction. Rule (2) moves the copy target, if a `let`-binding $x = C_1[y]$ is found: in this case, copying will first replace y in $C_1[y]$ instead of replacing x.

After the labeling is finished, the reduction rules are applied as follows (if possible): we only mention the redex with labels (outer contexts may be present):

$$
\begin{array}{ll}
(lbeta) & ((\lambda x.s)^\diamond\ t) \to \texttt{let}\ x = t\ \texttt{in}\ s \\
(cp) & \texttt{let}\ x = (\lambda y.s)^\diamond\ \texttt{in}\ C[x^\circledcirc] \to \texttt{let}\ x = \lambda y.s\ \texttt{in}\ C[\lambda y.s] \\
(llet) & \texttt{let}\ x = (\texttt{let}\ y = s\ \texttt{in}\ t)^\diamond\ \texttt{in}\ C[x^\circledcirc] \to \texttt{let}\ y = s\ \texttt{in}\ (\texttt{let}\ x = t\ \texttt{in}\ C[x])] \\
(lapp) & ((\texttt{let}\ x = s\ \texttt{in}\ t)^\diamond\ r) \to \texttt{let}\ x = s\ \texttt{in}\ (t\ r)
\end{array}
$$

Example 3.6.4. *The call-by-need evaluation of expression* $\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)\ x)$ *is as follows:*

$$
\begin{array}{rl}
 & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)\ x))^\star \\
\Rightarrow & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)\ x)^\star) \\
\Rightarrow & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^\diamond\ x)) \\
\xrightarrow{need,lbeta} & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))^\star \\
\Rightarrow & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)^\star) \\
\Rightarrow & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y^\star)) \\
\Rightarrow & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x^\diamond\ \texttt{in}\ y^\circledcirc)) \\
\Rightarrow & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x^\diamond\ \texttt{in}\ y)) \\
\Rightarrow & (\texttt{let}\ x = ((\lambda u.u)\ (\lambda w.w))^\diamond\ \texttt{in}\ (\texttt{let}\ y = x^\circledcirc\ \texttt{in}\ y)) \\
\Rightarrow & (\texttt{let}\ x = ((\lambda u.u)^\diamond\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^\circledcirc\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta} & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))^\star \\
\Rightarrow & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)^\star) \\
\Rightarrow & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y^\star))
\end{array}
$$

$$
\begin{array}{l}
\Rightarrow(\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y^{\circledcirc})) \\
\Rightarrow(\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y)) \\
\Rightarrow(\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}(\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)))^{\star} \\
\Rightarrow(\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))^{\star}) \\
\Rightarrow(\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)^{\star})) \\
\Rightarrow(\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y^{\star}))) \\
\Rightarrow(\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y^{\circledcirc}))) \\
\Rightarrow(\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y))) \\
\Rightarrow(\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y))) \\
\Rightarrow(\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\Rightarrow(\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)))^{\star} \\
\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))^{\star}) \\
\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)^{\star})) \\
\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y^{\star}))) \\
\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y^{\circledcirc}))) \\
\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y))) \\
\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ y)))^{\star} \\
\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ y))^{\star}) \\
\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ y)^{\star})) \\
\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ y^{\star}))) \\
\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)^{\diamond}\ \texttt{in}\ y^{\circledcirc}))) \\
\xrightarrow{need,cp}(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ (\lambda w.w))))
\end{array}
$$

FWHNFs of call-by-need evaluation are expressions of the form $LR[\lambda x.s]$, i.e. abstractions that are enclosed by a `let`-environment.

Definition 3.6.5. *An expression s* converges for call-by-need evaluation *(written as $s{\downarrow_{need}}$) iff it is call-by-need evaluated to an FWHNF, i.e.*

$$s{\downarrow_{need}} \iff \exists\, FWHNF\ v : s \xrightarrow{need,*} v$$

It is possible to show that convergence for call-by-need evaluation coincides with convergence for call-by-name evaluation (we omit the proof):

Proposition 3.6.6. *Let s be (*`let`*-free) expression, then $s\downarrow \iff s\downarrow_{need}$.*

3.7. Semantic Equality: Contextual Equivalence

We defined different operational semantics of the lambda calculus, and also defined (or implicitly used) three kinds of equality: Purely syntactic equality of expressions, equality up to α-renaming (i.e. α-equivalence), and β-convertibility, i.e. $\overset{C,\beta,*}{\longleftrightarrow}$. Of course, $\overset{C,\beta,*}{\longleftrightarrow}$ equates more expressions than $=_\alpha$, which again equates more expressions that syntactic equivalence. However, all of them are quite restricted and thus in this section we introduce a notion of semantic equivalence which is based on the operational semantics. Such a notion of equality can for instance be used to check whether the optimizations and transformations performed by a compiler are correct (i.e. if they preserve equality).

Leibniz' law of the identity of indiscernibles states that if objects o_1 and o_2 have the same property for all properties, then o_1 is identical to o_2. In every context, we can exchange o_1 by o_2 but no difference is observable. For program calculi like the lambda calculus the principle can be applied as follows: two expressions s and t are equal iff their behaviour cannot be distinguished independently in which context they are used. More formally, s and t are equal iff for all contexts C: the observable behaviours of $C[s]$ and $C[t]$ are the same. For deterministic languages, it is usually sufficient to observe the termination of programs (which we already defined as convergence, see Definitions 3.4.5, 3.5.3 and 3.6.5).

We define the *contextual equivalence* following this pattern, where we first define a contextual approximation and then the equivalence as symmetrization of the approximation.

Definition 3.7.1 (Contextual Approximation and Equivalence)**.** *For the call-by-name lambda calculus, we define the* contextual approximation $\leq_c$ *and* contextual equivalence $\sim_c$ *as:*

- $s \leq_c t$ *iff* $\forall C : C[s]\downarrow \implies C[t]\downarrow$
- $s \sim_c t$ *iff* $s \leq_c t$ *und* $t \leq_c s$

For the call-by-value lambda calculus, we define the contextual approximation $\leq_{c,value}$ *and* contextual equivalence $\sim_{c,value}$ *as:*

- $s \leq_{c,value} t$ *iff* $\forall C :$ *If* $C[s], C[t]$ *are closed and* $C[s]\downarrow_{value}$, *then also* $C[t]\downarrow_{value}$
- $s \sim_{c,value} t$ *iff* $s \leq_{c,value} t$ *and* $t \leq_{c,value} s$

For the call-by-need lambda calculus, we could give a similar definition, but we omit it.

In call-by-value calculi it is a difference iff all contexts C or only the closing contexts are used, while in call-by-name and call-by-need there is no difference.

A justification to use the closing ones is that they are the programs which are executed. Another one is that in call-by-value variables stand for values, while in call-by-name variables stand for any expression. For instance, the equation $x \sim_{c,value} \lambda y.(x\ y)$ holds (we omit the proof), but it would not hold if also non-closing contexts are used (consider the empty context: $x\Uparrow_{value}$, but $\lambda y.(x\ y)\downarrow_{value}$). The equation $x \sim_c \lambda y.(x\ y)$ does not hold, since the empty context distinguishes

both expressions, but also the closing context $C := (\lambda x.[\cdot])\ \Omega$ distinguishes the expression under call-by-name evaluation $C[x] \xrightarrow{name} \Omega \xrightarrow{name} \Omega \xrightarrow{name} \cdots$, but $C[\lambda y.(x\ y)] \xrightarrow{name} \lambda y.(\Omega\ y)$ which is an FWHNF. In general, the transformation $s \to \lambda x.(s\ x)$ is called *eta-expansion*, or in the other direction $\lambda x.(s\ x) \to s$ is called *eta-reduction*.

Contextual equivalence can be seen as the coarsest equivalence that distinguishes obviously different expressions. An important property of contextual equivalence is the following:

Proposition 3.7.2. *The contextual equivalences $\sim_c$ and $\sim_{c,value}$ are* congruences*, i.e. they are equivalence relations and compatible with contexts, i.e. $s \sim t \implies C[s] \sim C[t]$.*

The contextual approximations $\leq_c$ and $\leq_{c,value}$ are precongruences*, i.e. they are preorders and compatible with contexts, i.e. $s \leq t \implies C[s] \leq C[t]$.*

Proof. We first consider the call-by-name calculus and $\leq_c$. Since $C[s]\downarrow \implies C[s]\downarrow$, reflexivity of $\leq_c$ holds. For transitivity, let $r \leq_c s$ and $s \leq_c t$ and C be a context such that $C[r]\downarrow$. Then $C[s]\downarrow$ by $r \leq_c s$ and from $s \leq_c t$ we also have $C[t]\downarrow$. For compatibility, let $s \leq_c t$ and C be a context. Let C' be an arbitrary context such that $C'[C[s]]\downarrow$. Then $C'[C[t]]\downarrow$, since $C'[C[\cdot]]$ is also a context.

The reasoning to show that $\sim_c$ is a congruence follows by symmetry.

For the call-by-value lambda calculus, reflexivity of $\leq_{c,value}$ is straight-forward. For transitivity, let $r \leq_{c,value} s$ and $s \leq_{c,value} t$, and C be a context such that $C[r]$ and $C[t]$ are closed. If $C[s]$ is also closed, the reasoning is straight-forward. Otherwise, assume that $FV(C[s]) = \{x_1, \ldots, x_n\}$, let $v_1, \ldots, v_n$ be arbitrary closed values and $D = (\lambda x_1, \ldots, x_n.[\cdot])\ v_1\ \ldots\ v_n$. Since $C[r]$ and $C[t]$ are closed, $D[C[r]] \xrightarrow{value,*} C[r]$ and $D[C[t]] \xrightarrow{value,*} C[t]$. Thus, $D[C[r]]\downarrow_{value} \iff C[r]\downarrow_{value}$ and also $D[C[t]]\downarrow_{value} \iff C[t]\downarrow_{value}$.

Since $C[r]\downarrow_{value}$ and $r \leq_{c,value} s$, we have $D[C[s]]\downarrow$. From $s \leq_{c,value} t$, we have $D[C[t]]\downarrow_{value}$, and thus also $C[t]\downarrow_{value}$.

For compatibility, let $s \leq_{c,value} t$ and C be a context. Let C' be an arbitrary context such that $C'[C[s]]$ and $C'[C[t]]$ are closed and $C'[C[s]]\downarrow$. Then $C'[C[t]]\downarrow$, since $C'[C[\cdot]]$ is also a context and $C'[C[s]]$ and $C'[C[t]]$ are closed. Again, $\sim_{c,value}$ is a congruence by symmetry. □

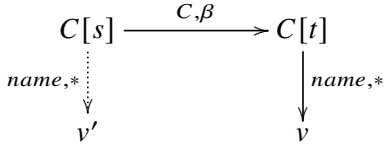
The congruence property allows to correctly transform subexpressions of larger expressions without considering the whole program: if the local transformation preserves contextual equivalence, the global programs are also contextually equivalent.

Contextual equivalence is a common notion for program equivalence used for several program calculi. A hurdle in using contextual equivalence is the universal quantification over all contexts. Disproving equivalences is easier, since a single contexts acting as a counter example is sufficient.

However, deciding whether two expressions are not contextually equivalent is undecidable (for instance, the question whether $s \not\sim_c \Omega$ is equivalent to the question if the program s terminates, and thus (since the lambda calculus is Turing complete) the halting problem is encodable, which is undecidable.)

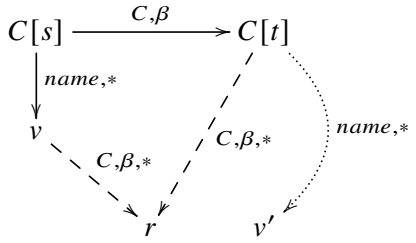
Proposition 3.7.3. *In the call-by-name lambda calculus (β) is a correct program transformation, i.e. if $s \xrightarrow{\beta} t$, then $s \sim_c t$.*

Proof. Let $s \xrightarrow{\beta} t$ and C be a context. If $C[t]\downarrow$, i.e. $C[t] \xrightarrow{name,*} v$ where v is an FWHNF, then the sequence $C[s] \xrightarrow{C,\beta} C[t] \xrightarrow{name,*} v$ can be used in Theorem 3.4.13, which shows $C[s]\downarrow$. This can be depicted as follows where straight lines are the given reductions and transformations, and the dotted ones follow from Theorem 3.4.13 and v, v' are FWHNFs.



Now let $C[s]\downarrow$, i.e. $C[s] \xrightarrow{name,n} v$ for some $n \in \mathbb{N}_0$ and an FWHNF v. This shows $v \xleftarrow{C,\beta} C[s] \xrightarrow{C,\beta} C[t]$, and thus Theorem 3.3.13 shows that there exists an expression r, s.t. $v \xrightarrow{C,\beta,*} r \xleftarrow{C,\beta,*} C[t]$. Since v is an FWHNF, r must be an FWHNF too, since $\xrightarrow{C,\beta}$-steps cannot transform an abstraction into an expression that is not an abstraction. Thus, Theorem 3.4.13 can be applied to $C[t] \xrightarrow{C,\beta,*} r$ which shows that $C[t]\downarrow$.

This can be depicted as follows where straight lines are the given reductions and transformations, the dashed ones follow from Theorem 3.3.13 and the dotted ones follow from Theorem 3.4.13 and v, r, v' are FWHNFs.



□

3.8. The Context Lemma

To restrict the number of contexts that need to be considered for proving that two expressions are contextually equal, one can try to prove a context lemma: the idea is to show that considering a subset of the contexts is sufficient.

We only consider the call-by-name lambda calculus, for the call-by-value lambda calculus, similar results can be obtained. For proving the context lemma, we require so-called multi-contexts: these are expressions with several (or no) holes at expression position. We assume that the holes

are numbered and write $M[\cdot_1, \ldots, \cdot_n]$ for a multi-context with n holes and $M[s_1, \ldots, s_n]$ for the expression where hole $\cdot_i$ of M is replaced by expression s_i (for $i = 1, \ldots, n$).

Lemma 3.8.1 (Context Lemma)**.** *Let s and t be closed expressions. If for all reductions contexts R, the implication $R[s]{\downarrow} \implies R[t]{\downarrow}$ holds, then also $s \leq_c t$ holds.*

Proof. We prove the more general claim using multi-contexts:

> If for all closed expressions s_i, t_i and for $i = 1, \ldots, n$: for all reduction contexts R the implication $R[s_i]{\downarrow} \implies R[t_i]{\downarrow}$ holds, then for all multi-contexts M the implication $M[s_1, \ldots, s_n]{\downarrow} \implies M[t_1, \ldots, t_n]{\downarrow}$ holds.

Assume that the preconditions hold, and that $M[s_1, \ldots, s_n]{\downarrow}$, i.e. $M[s_1, \ldots, s_n] \xrightarrow{name,m} v$ where v is an FWHNF. We use induction on the following pair, ordered lexicographically:

1. The number m of call-by-name reductions from $M[s_1, \ldots, s_n]$ to an FWHNF.
2. The number n of holes of M.

As a base case, consider the case that $n = 0$ then M is an expression and the claim holds (since $M{\downarrow}$ independently of any expressions s_i, t_i). This base case includes the case $(m, n) = (0, 0)$.

For the induction step, assume that $n > 0$:

We make a case distinction:

- A hole of M is a reduction context, i.e. $M[s_1, \ldots, s_{i-1}, \cdot_i, s_{i+1}, \ldots, s_n]$. Then there exists a hole j, such that $M[r_1, \ldots, r_{j-1}, \cdot_j, r_{j+1}, \ldots, r_n]$ is a reduction context for any expressions $r_1, \ldots, r_n$.

  Let $M' = M[\cdot_1, \ldots, \cdot_{j-1}, s_j, \cdot_{j+1}, \ldots, \cdot_n]$. Since $M'[s_1, \ldots, s_{j-1}, s_{j+1}, \ldots, s_n] = M[s_1, \ldots, s_n]$, both expressions have the same call-by-name reduction. Since the number of holes of M' is $n - 1$, we can apply the induction hypothesis, i.e. $M'[t_1, \ldots, t_{j-1}, t_{j+1}, t_n]{\downarrow}$. Since $C_s = M[s_1, \ldots, s_{j-1}, \cdot_j, s_{j+1}, \ldots, s_n]$ and $C_t = M[t_1, \ldots, t_{j-1}, \cdot_j, t_{j+1}, \ldots, t_n]$ are both reduction contexts and $M'[t_1, \ldots, t_{j-1}, t_{j+1}, t_n] = C_t[s_j]$, $C_t[s_j]{\downarrow}$ and the precondition shows that $C_t[t_j]{\downarrow}$. Since $C_t[t_j] = M[t_1, \ldots, t_n]$ this shows the claim.
- $n > 0$ and no hole is a reduction context. If $m = 0$, then $M[s_1, \ldots, s_n]$ is an FWHNF and $M[t_1, \ldots, s_n]$ must be an FWHNF too (in fact any $M[r_1, \ldots, r_n]$ is an FWHNF for any expressions $r_1, \ldots, r_n$).

  Otherwise, consider the reduction sequence $M[s_1, \ldots, s_n] \xrightarrow{name} s' \xrightarrow{name,m-1} v$ and inspect what can happen with the subexpressions $s_1, \ldots, s_n$ in M: Since no hole of M is in a reduction context, they can only change their position and may be duplicated or removed (if they are part of an argument that is substituted by the (β)-reduction). Since the expressions $s_1, \ldots, s_n$ are closed no other expression can be copied inside any s_i. Hence, there exists a multicontext M' with u holes, such that $s' = M'[s_{f(1)}, \ldots, s_{f(u)}]$ where $f : \{1, \ldots, u\} \to \{1, \ldots, n\}$. Moreover, $M[r_1, \ldots, r_n] \xrightarrow{name} M'[r_{f(1)}, \ldots, r_{f(u)}]$ for any expressions $r_1, \ldots, r_n$ and thus $M[t_1, \ldots, t_n] \xrightarrow{name} M'[t_{f(1)}, \ldots, t_{t(u)}] = t'$. Since

$s' \xrightarrow{name,m-1} v$ and the precondition holds for all pairs $s_{f(i)}, t_{f(i)}$ for $i = 1, \ldots, m$, we can apply the induction hypothesis to s' and t' showing $t'\downarrow$ and thus also $t\downarrow$. □

To prove equivalences for open expressions, the following proposition is helpful:

Proposition 3.8.2. *Let s and t be expressions with free variables $x_1, \ldots, x_n$ Then $s \leq_c t$ iff for all closed expressions $t_1, \ldots, t_n$: $s[t_1/x_1, \ldots, t_n/x_n] \leq_c t[t_1/x_1, \ldots, t_n/x_n]$*

Proof. If $s \leq_c t$, then $C[s] \leq_c C[t]$ for $C = \lambda x_1, \ldots, x_n.[\cdot]$. Since $C[s] \xrightarrow{C,\beta,*} s[t_1/x_1, \ldots, t_n/x_n]$, $C[t] \to t[t_1/x_1, \ldots, t_n/x_n]$ and (β) is correct (Proposition 3.7.3), $s[t_1/x_1, \ldots, t_n/x_n] \leq_c t[t_1/x_1, \ldots, t_n/x_n]$.

For the other direction, we use induction on the number n of free variables of s and t. It is also clear, that it suffices to show that $\lambda x_1, \ldots, x_n.s \leq_c \lambda x_1, \ldots, x_n.t$, since then the context $C := [\cdot]\ x_1\ \ldots\ x_n$ shows $s \leq_c t$.

If $n = 0$, then the claim holds.

If $n > 0$, we use the context lemma, and show $R[\lambda x_1, \ldots, x_n.s]\downarrow \implies R[\lambda x_1, \ldots, x_n.t]\downarrow$: for all reduction contexts R. Let R be a reduction context, and $R[\lambda x_1, \ldots, x_n.s]\downarrow$. If R is the empty context, then both expressions are FWHNFs and the claim holds. If $R = R'[[\cdot]\ r]$ for some reduction context R', then $R[\lambda x_1, \ldots, x_n.s] \xrightarrow{name} R'[\lambda x_2, \ldots, x_n.s[r/x_1]]$ and $R[\lambda x_1, \ldots, x_n.t] \xrightarrow{name} R'[\lambda x_2, \ldots, x_n.t[r/x_1]]$ Then from the given condition we have $s[r/x_1][t_2/x_2, \ldots, t_n/x_n] \leq_c t[r/x_1][t_2/x_2, \ldots, t_n/x_n]$ for all expressions $t_2, \ldots, t_n$. Since $s[r/x_1]$ and $t[r/x_1]$ have only $n-1$ free variables, the induction hypothesis shows $\lambda x_2, \ldots, x_n.s[r/x_1] \leq_c \lambda x_2, \ldots, x_n.t[r/x_1]$. Since $\leq_c$ is a precongruence, this shows $R[\lambda x_2, \ldots, x_n.s[r/x_1]] \leq_c R[\lambda x_2, \ldots, x_n.t[r/x_1]]$. Applying correctness of (β) shows $R[\lambda x_1, \ldots, x_n.t]\downarrow$ □

For the call-by-value lambda calculus, $(\beta_{value}) \subseteq \sim_{c,value}$ holds (we do not provide the proof), but $(\beta) \not\subseteq \sim_{c,value}$: For example, $((\lambda x.(\lambda y.y))\ \Omega)\Uparrow_{value}$ and $\lambda y.y\downarrow_{value}$, thus the expressions are different (where the empty context is a counter-example).

The contextual equivalences w.r.t. call-by-name and call-by-value evaluation are not related, i.e. $\sim_c \not\subseteq \sim_{c,value}$ and $\sim_{c,value} \not\subseteq \sim_c$. The first part follows from correctness of (β), the second part can be shown for instance by verifying that $((\lambda x.(\lambda y.y))\ \Omega) \sim_{c,value} \Omega$ but $((\lambda x.(\lambda y.y))\ \Omega) \not\sim_c \Omega$.

For the contextual approximation there are least and greatest elements in both lambda calculi:

Proposition 3.8.3. *All closed diverging expressions are least elements w.r.t. $\leq_c$ and $\leq_{c,value}$. For instance $\Omega \leq_c s$ and also $\Omega \leq_{c,value} s$ for all expressions s.*

With $K := \lambda x.\lambda y.x$, $Y := \lambda f.(\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))$, and $Z := \lambda f.(\lambda x.(f\ \lambda z.(x\ x)\ z))\ (\lambda x.(f\ \lambda z.(x\ x)\ z))$ (see Example 3.1.3) the expression $Y\ K$ is a greatest element of $\leq_c$ and $Z\ K$ is a greatest element of $\leq_{c,value}$, i.e. $s \leq_c Y\ K$ and $s \leq_c Z\ K$ for all expressions s.

Proof sketch. We omit the proofs for the call-by-value lambda calculus and only consider the call-by-name part of the proposition. Let $\bot$ be a closed diverging expression and s be an arbitrary closed expression. Let R be an arbitrary reduction context, then $R[\bot]$ cannot converge, i.e. $R[\bot]\Uparrow$. The context lemma now immediately shows $\bot \leq_c s$. Since $\bot$ is closed, Proposition 3.8.2 shows $\bot \leq_c s$ for any (perhaps also open) expression s

For analyzing $(Y\ K)$, let $r_y = (\lambda x.K\ (x\ x))$. Then $Y\ K \xrightarrow{C,\beta} r_y\ r_y \xrightarrow{C,\beta} K\ (r_y\ r_y)$ and thus $(Y\ K)\ s_1\ \ldots\ s_n \xrightarrow{C,\beta,*} K\ (r_y\ r_y)\ s_1\ \ldots\ s_n \xrightarrow{C,\beta,*} (r_y\ r_y)\ s_2\ \ldots\ s_n \xrightarrow{C,\beta,*} (r_y\ r_y) \xrightarrow{C,\beta} K\ (r_y\ r_y) \xrightarrow{C,\beta} \lambda x.(r_y\ r_y)$

Using Theorem 3.4.13, this shows that for every reduction context R: $R[(Y\ K)]\downarrow$. Since $(Y\ K)$ is a closed expression, the context lemma shows that $s \leq_c (Y\ K)$ for every closed expression s. Since $(Y\ K)$ is closed, Proposition 3.8.2 shows $s \leq_c (Y\ K)$ for any (perhaps also open) expression s. □

Remark 3.8.4. *We explain the call-by-value evaluation of $(Z\ K)$: With $r_z = (\lambda x.(K\ \lambda z.(x\ x)\ z))$, one can verify that $Z\ K \xrightarrow{value} r_z\ r_z \xrightarrow{value} K\ \lambda z.((r_z\ r_z)\ z) \xrightarrow{value} \lambda y.\lambda z.(r_z\ r_z)\ z$ and thus for values $v_1, \ldots, v_n$: $(Z\ K)\ v_1\ \ldots\ v_n \xrightarrow{value,*} (r_z\ r_z)\ v_1\ \ldots\ v_n \xrightarrow{value,*} (\lambda y.\lambda z.(r_z\ r_z)\ z)\ v_1\ \ldots\ v_n \xrightarrow{value} (\lambda z.(r_z\ r_z)\ z)\ v_2 \ldots\ v_n \xrightarrow{value} (r_z\ r_z)\ v_2\ v_3 \ldots\ v_n \xrightarrow{value,*} (r_z\ r_z) \xrightarrow{value,*} \lambda y.\lambda z.(r_z\ r_z)\ z$*

3.9. Turing Completeness of the Lambda-Calculus

Instead of providing a formal proof of the Turing completeness of the lambda calculus, we only argue that it is Turing complete. A formal proof can be found in (Hankin, 2004), where it is shown that μ-recursive functions can be expressed in the lambda calculus. For instance, in (Schöning, 2008) a proof can be found that μ-recursive functions are expressive as Turing machines (i.e. are Turing complete).

A direct proof to show Turing completeness is by simulating a Turing machine in the lambda calculus. In Chapter A we provide a simulation of Turing machines in the functional programming language Haskell. This shows that Haskell is Turing complete. We compare the used constructs in Haskell and the lambda calculus to see what is missing and how this could be resolved in a proof of Turing completeness of the lambda calculus.

Named function definitions: A Haskell-function

$$f\ x_1\ \ldots\ x_n = e$$

can be represented by the lambda abstraction

$$\lambda x_1, \ldots, x_n.e$$

as long as e does not call f (or other functions that again call f), i.e. f must not be recursive for this approach. For recursive functions the fixpoint combinator can be used to represent recursion. For simplicity, let us assume, that e only calls f, but no other functions. Then f can be encoded by

$$Y\ (\lambda f.\lambda x_1, \ldots, x_n.e).$$

We inspect the behaviour: Let $F = (\lambda f.\lambda x_1, \ldots, x_n.e)$ and $r_y = (\lambda x.F\ (x\ x))$. Then

$$Y\ F \xrightarrow{C,\beta} r_y\ r_y = (\lambda x.F\ (x\ x))\ r_y \xrightarrow{C,\beta} F\ (r_y\ r_y) \xleftarrow{C,\beta} F\ (Y\ F),$$

i.e. $Y\ F \sim_c F\ (Y\ F)$. This shows $Y\ F \sim_c F^i\ (Y\ F)$ where F^i is the i-fold application of F, and also $Y\ F \sim_c F\ (Y\ F) \sim_c \lambda x_1, \ldots, x_n.e[(Y\ F)/f]$. For mutual recursive functions, the encoding is a bit more complicated, but still possible.

Data: Besides some syntactic sugar (like the record syntax), the Haskell-program uses data, data types and constructors and selectors to construct and deconstruct data. It is quite obvious, that it should be sufficient to represent booleans, tuples, lists of arbitrary length and natural numbers to encode the program (together with selectors to e.g. select the first component of a pair). However, the lambda calculus has none of these constructs. But, they can be encoded as lambda expressions (there are different encodings). The following encoding sketches the so-called Church-encoding of numbers, booleans, pairs and lists:

The idea is that number i is represented by the i-fold function composition, i.e. for any unary function f, f represents 1, $f \circ f$ represents 2, $f \circ f \circ f$ represents 3 and so on, 0 is represented by the identity.

Each number is a lambda expression that takes 2 parameters. The encoding is $0 := \lambda f.\lambda x.x$ and for $i > 0$: $i := \lambda f.\lambda x.f^i\ x$

The addition for instance takes two such functions and combines them to a new one, using the identity $f^m \circ f^n = f^{m+n}$:

$$plus = \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x).$$

The successor of a number can be computed in the same way where m is 1:

$$succ = \lambda n.\lambda f.\lambda x.f\ (n\ f\ x).$$

The predecessor is complicated, we do not explain the encoding, but provide it (it behaves as follows $pred\ 0 = 0$ and $pred\ i = i - 1$ for $i > 0$). The encoding is

$$pred = \lambda n.\lambda f.\lambda x.n\ (\lambda g.\lambda h.h\ (g\ f))\ (\lambda z.x)\ (\lambda u.u).$$

Booleans are defined as $true = \lambda x.\lambda y.x$ and $false = \lambda x.\lambda y.y$. Note that $b\ s\ t$ for $b \in \{true, false\}$ behaves like `if` b `then` s `else` t.

Pairs can be encoded as

$$pair = \lambda x.\lambda y.\lambda z.z\ x\ y.$$

The first two arguments are the arguments of the pair, the third one is for the selector. Selector function can be defined as

$$\begin{array}{lcl} first & = & \lambda p.p\ K \\ second & = & \lambda p.p\ K_2 \end{array}$$

With these definitions $first\ (pair\ s\ t) \sim_c s$ and $second\ (pair\ s\ t) \sim_c t$.

Non-empty lists can be encoded by pairs p where the first component of p is the head element of the list, and the second component of p is the remaining list (i.e. the tail of the list). However, the empty list cannot be encoded in this way. This can be corrected by using pairs $pair\ flag\ p$ where *flag* is $true$ or $false$ and $pair\ true\ s$ means the empty list (independent of s) and $pair\ false\ s$ is a non-empty list. Then we can use the encodings:

$$\begin{array}{lcl} nil & = & pair\ true\ true \\ cons & = & \lambda h.\lambda t.pair\ false\ (pair\ h\ t) \end{array}$$

Then we may define a test for emptiness and selector functions:

$$\begin{array}{lcl} isNil & = & first \\ head & = & \lambda l.first\ (second\ l) \\ tail & = & \lambda l.second\ (second\ l) \end{array}$$

This shows that data can be encoded using pure lambda expressions. Disadvantages are that one cannot distinguish between data and functions (and in case of the provided encoding also not between different data types, since e.g. the encoding of 0 and the encoding of $false$ is the same.)

Types: Haskell only permits well-typed programs, while the lambda calculus has no types. But this is not a restriction for showing Turing completeness, since the typed encoding (in form of a Haskell program) can also be used as an untyped one (doing it in the other way, would be problem).

However, even we now know that the lambda calculus is Turing complete and thus it could be used to reason about functional programs as a core language, it is very difficult to express everything with abstractions and applications and without types. Moreover, since different Haskell-programs would be mapped to the same lambda expression (e.g. 0 and False), equivalences proved in the lambda calculus would not necessarily hold for Haskell-programs.

One more difference is, that Haskell has a `seq`-operator which mimics strict (i.e. call-by-value) evaluation, which is not expressible in the pure lambda calculus with call-by-name evaluation.

For all these reasons we will consider extended core languages which fit better as a core language for Haskell and other functional programming languages.

3.10. Conclusion and References

We introduced the lambda calculus, different evaluation strategies and the concept of contextual semantics. A lot of resources on the lambda calculus exist, we mention some of them. A standard reference (often not easy to understand) is (Barendregt, 1984). A good introduction can be found in (Hankin, 2004). Call-by-need lambda calculi with `let` were introduced in (Ariola et al., 1995; Ariola & Felleisen, 1997) and (Maraist et al., 1998). Non-deterministic extensions are in (Kutzner & Schmidt-Schauß, 1998; Kutzner, 2000; Mann, 2005b; Mann, 2005a). The call-by-need reduction in this chapter is mainly taken from (Mann, 2005a). We did non consider call-by-need lambda calculi with recursive `let`, they are treated for instance in (Schmidt-Schauß et al., 2008; Schmidt-Schauß et al., 2010). The notion of contextual equivalence dates back to (Morris, 1968). In the context of the lambda calculus it was for instance investigated in (Plotkin, 1975). A lot of references and discussions on the context lemma can be found in (Schmidt-Schauß & Sabel, 2010).

4. Core Languages of Non-Strict Functional Programming

In this chapter we modularly extend the call-by-name lambda calculus with constructs that better match the core language of non-strict functional programming languages such as Haskell. We always compare our core language with Haskell and see what is missing. First we add data constructors and case expressions, then we add recursive function definitions, and before considering polymorphic typing, we add Haskell's `seq`-operator. In this way, we introduce several core languages. These core languages are mainly from (Schmidt-Schauß, 2009).

4.1. The Core Language KFPT

4.1.1. Syntax of KFPT

As a first extension we add data to the lambda calculus. We do this by adding *data constructors* c_i that, if applied to arguments, represent data (for example, c_i might be a *pair* that is applied to two arguments, and thus *pair s t* is a pair.) To have a common mechanism and notion for selecting parts of the data (e.g. the first argument of a pair), we use `case`-expressions. To overload the notation, we introduce a very weak form of typing: We assume that the set of all data constructors is partitioned into subsets, where each subset represents a type and the set has a name: for example, such sets are `Bool`, `List`, `Pair`, ... All sets are finite, so each type T has a finite number of data constructors, each written as c_i for some number i. Each constructor c_i has a fixed *arity* $\mathrm{ar}(c) \in \mathbb{N}_0$. For some types, we use some better names for the constructor to match the Haskell wording and notation: The type `Bool` has constructors `True` and `False` both of arity 0. The type `Pair` has a single constructor `Pair` of arity 2, sometimes we write (s, t) instead of `Pair` *s t*. The type `List` has constructors `Nil` (of arity 0) and `Cons` of arity 2. The constructor `Nil` represents the empty list, and for example, a list consisting of boolean values true, false, true is written as `Cons True` (`Cons False` (`Cons True Nil`)). We also use Haskell notation for lists and write : instead of `Cons` (and written infix instead of prefix) and [] instead of `Nil`, e.g. we write `True`:`False`:`True`: []. As syntactic sugar we also write [`True`,`False`,`True`] for the same list.

We assume that data constructors only occur fully saturated, i.e. if $\mathrm{ar}(c_i) = n$, then c_i only occurs with n arguments.

[1] The list constructor : is right-associative, i.e. $a_1 : a_2 : a_3 : []$ means $a_1 : (a_2 : (a_3 : []))$.

Definition 4.1.1. *The syntax of the* core language KFPT *is generated by the following grammar (and side-conditions) with start non-terminal* **Expr***, where V, V_i are variables.*

$$
\begin{array}{rcl}
\mathbf{Expr} & ::= & V \mid \lambda V.\mathbf{Expr} \mid (\mathbf{Expr}_1\ \mathbf{Expr}_2) \\
& \mid & (c_i\ \mathbf{Expr}_1\ \ldots\ \mathbf{Expr}_{\mathrm{ar}(c_i)}) \\
& \mid & (\mathtt{case}_T\ \mathbf{Expr}\ \mathtt{of}\ \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_n \to \mathbf{Expr}_n\}) \\
& & \quad \textit{Here } \mathbf{Pat}_i \textit{ is a pattern for constructor } c_i, \\
& & \quad \mathbf{Pat}_i \to \mathbf{Expr}_i \textit{ is called a } \mathtt{case}\textit{-alternative.} \\
& & \quad \textit{For every constructor of type } T\textit{, there} \\
& & \quad \textit{is exactly one } \mathtt{case}\textit{-alternative.}
\end{array}
$$

$$
\mathbf{Pat}_i \quad ::= \quad (c_i\ V_1 \ldots V_{ar(c_i)}) \textit{ where the variables } V_i \textit{ are pairwise distinct.}
$$

Note that the `case` *is labeled with the type as a subscript and the side conditions ensure that for each constructor of the corresponding type, there is exactly one* `case`*-alternative in the set of alternatives.*

Except for the weak typing by labeling the `case` and partitioning the data constructors, there is no typing for KFPT. So KFPT is weakly typed, extends the lambda calculus by *constructor applications* $(c_i\ s_1\ \ldots\ s_{\mathrm{ar}(c_i)})$ and `case`-expressions.

Compared to Haskell, the `case`-expressions of KFPT are more restrictive, since for all constructors there must be an alternative (in Haskell a missing alternative can lead to a runtime error), patterns are flat (nested patterns like $(x : (y : ys))$ are not allowed in KFPT), and there are no default alternatives like in Haskell (they match if no other alternative matches).

However, a complex Haskell `case`-expression can be transformed into multiple `case`-expressions that match the syntax of KFPT: Nested patterns are translated into nested `case`-expressions and missing alternatives are added, where the right-hand side is a closed non-terminating expression such as Ω.

As an abbreviation we sometimes write $(\mathtt{case}_T\ s\ \mathtt{of}\ Alts)$ where $Alts$ is a placeholder for (syntactically correct) `case`-alternatives.

Example 4.1.2. *Projections fst and snd, which compute the first or the second component of a pair, can be implemented in* KFPT *as abstractions:*

$$
\begin{array}{rcl}
fst & := & \lambda x.\mathtt{case}_{\mathtt{Pair}}\ x\ \mathtt{of}\ \{(\mathtt{Pair}\ a\ b) \to a\} \\
snd & := & \lambda x.\mathtt{case}_{\mathtt{Pair}}\ x\ \mathtt{of}\ \{(\mathtt{Pair}\ a\ b) \to b\}
\end{array}
$$

A function that computes the first element of a list can be expressed in KFPT *as:*

$$
\lambda xs.\mathtt{case}_{\mathtt{List}}\ xs\ \mathtt{of}\ \{\mathtt{Nil} \to \bot; (\mathtt{Cons}\ y\ ys) \to y\}
$$

Similarly, the following function computes the tail of a list:

$$
\lambda xs.\mathtt{case}_{\mathtt{List}}\ xs\ \mathtt{of}\ \{\mathtt{Nil} \to \bot; (\mathtt{Cons}\ y\ ys) \to ys\}
$$

Testing whether a list is empty or not, can be programmed as follows:

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{True}; (\texttt{Cons}\ y\ ys) \to \texttt{False}\}$$

In Haskell there are `if-then-else`*-expressions of the form* `if` e `then` s `else` t*. In* KFPT *they can be simulated by:*

$$\texttt{case}_{\texttt{Bool}}\ e\ \texttt{of}\ \{\texttt{True} \to s; \texttt{False} \to t\}$$

Compared to the lambda calculus, KFPT has one new construct that *binds* variables: in a `case`-alternative $(c_i\ x_1\ \ldots\ x_{\mathrm{ar}(c_i)}) \to s$, the variables $x_1, \ldots, x_{\mathrm{ar}(c_i)}$ are bound with scope s. Formally, the sets of *free variables* $FV(s)$ and *bound variables* $BV(s)$ for an expression s are defined as:

$$\begin{array}{ll}
FV(x) & = x \\
FV(\lambda x.s) & = FV(s) \setminus \{x\} \\
FV(s\ t) & = FV(s) \cup FV(t) \\
FV(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)}) & = FV(s_1) \cup \ldots \cup FV(s_{\mathrm{ar}(c_i)}) \\
FV(\texttt{case}_T\ t\ \texttt{of} & = FV(t) \cup (\bigcup_{i=1}^{n} (FV(s_i) \setminus \{x_{i,1}, \ldots, x_{i,\mathrm{ar}(c_i)}\})) \\
\quad \{(c_1\ x_{1,1}\ \ldots\ x_{1,\mathrm{ar}(c_1)}) \to s_1; & \\
\quad \ldots & \\
\quad (c_n\ x_{n,1}\ \ldots\ x_{n,\mathrm{ar}(c_n)}) \to s_n\}) &
\end{array}$$

$$\begin{array}{ll}
BV(x) & = \emptyset \\
BV(\lambda x.s) & = BV(s) \cup \{x\} \\
BV(s\ t) & = BV(s) \cup BV(t) \\
BV(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)}) & = BV(s_1) \cup \ldots \cup BV(s_{\mathrm{ar}(c_i)}) \\
BV(\texttt{case}_T\ t\ \texttt{of} & = BV(t) \cup (\bigcup_{i=1}^{n} (BV(s_i) \cup \{x_{i,1}, \ldots, x_{i,\mathrm{ar}(c_i)}\})) \\
\quad \{(c_1\ x_{1,1}\ \ldots\ x_{1,\mathrm{ar}(c_1)}) \to s_1; & \\
\quad \ldots & \\
\quad (c_n\ x_{n,1}\ \ldots\ x_{n,\mathrm{ar}(c_n)}) \to s_n\}) &
\end{array}$$

As in the lambda calculus, an expression s is *closed*, if $FV(s) = \emptyset$, and otherwise it is *open*. Renaming of bound variables is called α-renaming, which allows variables to be consistently renamed at binders and in their scope. We omit the formal definition. As in the lambda calculus we use the distinct variable convention (see Definition 3.2.1) and α-renaming can be used to follow the convention.

Example 4.1.3. *For the* KFPT*-expression*

$$s := ((\lambda x.\texttt{case}_{\texttt{List}}\ x\ \texttt{of}\ \{\texttt{Nil} \to x; \texttt{Cons}\ x\ xs\ \to \lambda u.(x\ \lambda x.(x\ u))\})\ x)$$

we have

$$
\begin{aligned}
& FV(s) \\
=& (FV(\lambda x.\mathtt{case}_{\mathtt{List}}\ x\ \mathtt{of}\ \{\mathtt{Nil} \to x; \mathtt{Cons}\ x\ xs\ \to \lambda u.(x\ \lambda x.(x\ u)))) \cup FV(x) \\
=& (FV(\lambda x.\mathtt{case}_{\mathtt{List}}\ x\ \mathtt{of}\ \{\mathtt{Nil} \to x; \mathtt{Cons}\ x\ xs\ \to \lambda u.(x\ \lambda x.(x\ u)))) \cup \{x\} \\
=& (FV(\mathtt{case}_{\mathtt{List}}\ x\ \mathtt{of}\ \{\mathtt{Nil} \to x; \mathtt{Cons}\ x\ xs\ \to \lambda u.(x\ \lambda x.(x\ u))) \setminus \{x\}) \cup \{x\} \\
=& ((FV(x) \cup (FV(x) \setminus \emptyset) \cup (FV(\lambda u.(x\ \lambda x.(x\ u))) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup (\{x\} \setminus \emptyset) \cup (FV(\lambda u.(x\ \lambda x.(x\ u))) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup (FV(\lambda u.(x\ \lambda x.(x\ u))) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((FV(x\ \lambda x.(x\ u)) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((FV(x) \cup FV(\lambda x.(x\ u)) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((\{x\} \cup FV(\lambda x.(x\ u)) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((\{x\} \cup (FV(x\ u) \setminus \{x\}) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((\{x\} \cup (FV(x) \cup FV(u)) \setminus \{x\}) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((\{x\} \cup (\{x\} \cup \{u\}) \setminus \{x\}) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((\{x\} \cup (\{x, u\} \setminus \{x\}) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((\{x\} \cup \{u\} \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup (\{x\} \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup \emptyset) \setminus \{x\}) \cup \{x\} \\
=& \{x\}.
\end{aligned}
$$

and $BV(s) = \{x, xs, u\}$ (we omit the computation), i.e. variable x occurs both bound and free and thus s violates the distinct variable convention. After α-renaming of all bound variables, we derive

$$s' := ((\lambda x_1.\mathtt{case}_{\mathtt{List}}\ x_1\ \mathtt{of}\ \{\mathtt{Nil} \to x_1; \mathtt{Cons}\ x_2\ xs\ \to \lambda u.(x_2\ \lambda x_3.(x_3\ u))\})\ x)$$

Now, the expression s' fulfills the distinct variable convention and $FV(s') = \{x\}$ and $BV(s') = \{x_1, x_2, xs, x_3, u\}$.

4.1.2. Operational Semantics of KFPT

We define call-by-name evaluation for KFPT, where $s[t/x]$ is the expression s where all free occurrences of x are replaced by t, and $s[t_1/x_1, \ldots, t_n/x_n]$ is the parallel substitution of x_i by t_i (for $i = 1, \ldots, n$) in expression s.

Definition 4.1.4. *The reduction rules* (β) *and* $(case)$ *in* KFPT *are defined as:*

$$\begin{array}{ll} (\beta) & (\lambda x.s)\ t \xrightarrow{\beta} s[t/x] \\ (case) & \texttt{case}_T\ (c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ \texttt{of}\ \{\ldots;\ (c\ x_1 \ldots\ x_{\mathrm{ar}(c)}) \to t;\ \ldots\} \\ & \quad \xrightarrow{case} t[s_1/x_1, \ldots, s_{\mathrm{ar}(c)}/x_{\mathrm{ar}(c)}] \end{array}$$

The (β)-rule is the same as in the lambda calculus. The (case)-rule evaluates a `case`-expression: if the first argument is a constructor application of the correct type, then the right-hand side of the matching alternative is used, with the given arguments replacing the formal parameters of the pattern.

Example 4.1.5. *The expression*

$$(\lambda x.\texttt{case}_{\texttt{Pair}}\ x\ \texttt{of}\ \{(\texttt{Pair}\ a\ b \to a\})\ (\texttt{Pair}\ \texttt{True}\ \texttt{False})$$

can be transformed into `True` *by* (β) *and* $(case)$ *reductions:*

$$\begin{array}{ll} & (\lambda x.\texttt{case}_{\texttt{Pair}}\ x\ \texttt{of}\ \{(\texttt{Pair}\ a\ b) \to a\})\ (\texttt{Pair}\ \texttt{True}\ \texttt{False}) \\ \xrightarrow{\beta} & \texttt{case}_{\texttt{Pair}}\ (\texttt{Pair}\ \texttt{True}\ \texttt{False})\ \texttt{of}\ \{(\texttt{Pair}\ a\ b) \to a\} \\ \xrightarrow{case} & \texttt{True} \end{array}$$

If $r_1 \to r_2$ using (β)- or (case)-reductions, then r_1 *directly reduces* to r_2. Contexts are KFPT-expressions that have a hole $[\cdot]$ at expression position, i.e. they are defined by the following grammar:

$$\begin{array}{rl} \textbf{Ctxt} ::= & [\cdot] \mid \lambda V.\textbf{Ctxt} \mid (\textbf{Ctxt}\ \textbf{Expr}) \mid (\textbf{Expr}\ \textbf{Ctxt}) \\ \mid & (c_i\ \textbf{Expr}_1\ \ldots \textbf{Expr}_{i-1}\ \textbf{Ctxt}\ \textbf{Expr}_{i+1}\ \textbf{Expr}_{\mathrm{ar}(c_i)}) \\ \mid & (\texttt{case}_T\ \textbf{Ctxt}\ \texttt{of}\ \{\textbf{Pat}_1 \to \textbf{Expr}_1; \ldots; \textbf{Pat}_n \to \textbf{Expr}_n\}) \\ \mid & (\texttt{case}_T\ \textbf{Expr}\ \texttt{of}\ \{\textbf{Pat}_1 \to \textbf{Expr}_1; \ldots; \textbf{Pat}_i \to \textbf{Ctxt}; \ldots, \textbf{Pat}_n \to \textbf{Expr}_n\}) \end{array}$$

As before, $C[s]$ is the expression derived by replacing the hole of C with the expression s. If a (β)- or a (case)-reduction is applied in a context C, i.e. $C[s] \to C[t]$ where s directly reduces to t, then the subexpression s in $C[s]$ (with its position at the hole of C) is called the *redex* (reducible expression) of the reduction step $C[s] \to C[t]$.

To define the call-by-name evaluation, we define reduction contexts:

Definition 4.1.6. *Reduction contexts R in* KFPT *are built by the following grammar with start symbol* **RCtxt***:*

$$\textbf{RCtxt} ::= [\cdot] \mid (\textbf{RCtxt}\ \textbf{Expr}) \mid (\texttt{case}_T\ \textbf{RCtxt}\ \texttt{of}\ \mathit{Alts})$$

Call-by-name-reduction in KFPT is defined as follows:;

Definition 4.1.7. *If s reduces directly to t and R is a reduction context, then $R[s] \xrightarrow{name} R[t]$ is a* call-by-name reduction *in* KFPT.

To denote the reduction rule used, we also write $\xrightarrow{name,\beta}$ or $\xrightarrow{name,case}$, respectively. We also write $\xrightarrow{name,+}$ for the transitive and $\xrightarrow{name,*}$ for the reflexive-transitive closure of $\xrightarrow{name}$.

Example 4.1.8. *The reduction $(\lambda x.x)\ ((\lambda y.y)\ (\lambda z.z)) \xrightarrow{name} (\lambda y.y)\ (\lambda z.z)$ is a call-by-name reduction. The reduction $(\lambda x.x)\ ((\lambda y.y)\ (\lambda z.z)) \to (\lambda x.x)\ (\lambda z.z)$ is* not *a call-by-name reduction, because the context $(\lambda x.x)\ [\cdot]$ is* not *a reduction context.*

The call-by-name evaluation is a sequence of call-by-name reduction steps. It ends successfully if a WHNF (weak head normal form) is reached. Let us define some kinds of normal forms

Definition 4.1.9. *A* KFPT*-expression s is a*

- normal form *(NF), if s does not contain any (β)- or $(case)$-redex.*
- head normal form *(HNF), if s is a constructor application or an abstraction $\lambda x_1, \ldots x_n.s'$ where s' is either a variable, a constructor application or of the form $(x\ s'')$ is (where x is a variable).*
- functional weak head normal form *(FWHNF) if s is an abstraction.*
- constructor weak head normal form *(CWHNF) if s is a constructor application $(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})$.*
- weak head normal form *(WHNF), if s is an FWHNF or a CWHNF.*

Note that every normal form is also head normal form, but a head normal form is not always a normal form, since in the arguments (of constructors or applications $(x\ s)$) a head normal form may contain redexes. Every HNF is also a WHNF (but not vice versa). We are mainly interested in WHNFs.

Definition 4.1.10 (Convergence)**.** *A* KFPT*-expression s* converges *(or* terminates*, written as $s{\downarrow}$) iff it can be evaluated using call-by-name evaluation to a WHNF, i.e.*

$$s{\downarrow} \iff \exists\ \mathit{WHNF}\ t : s \xrightarrow{name,*} t$$

If s does not converge, then we say s diverges *and write $s{\Uparrow}$.*

We also say that *s has a WHNF* (FWHNF, CWHNF, resp.) if s can be evaluated to a WHNF (FWHNF, CWHNF, resp.) using a finite number of call-by-name reductions.

4.1.3. Dynamic Typing

Call-by-name evaluation may stop (get stuck, i.e. no more $\xrightarrow{name}$-reduction is applicable) *without* reaching a WHNF. As in the lambda calculus this can happen if a free variable occurs at reduction position, i.e. the expression is of the form $R[x]$ where R is a reduction contexts and x is a variable.

However, unlike the lambda calculus, evaluation may get stuck, because a type error has been detected (because abstractions are detected at a position where data is required, or because the data given is of a different type than the required type). Since type errors are detected during evaluation, this is called a *dynamic* type error.

Definition 4.1.11 (Dynamic Typing Rules for KFPT)**.** *Let s be a* KFPT*-expression. We say that s is* directly dynamically untyped*, if s is of one of the following forms (where R is a reduction context):*

- $R[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ Alts]$ *and c is* not *of type T.*
- $R[\texttt{case}_T\ \lambda x.t\ \texttt{of}\ Alts]$.
- $R[(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ t]$

A KFPT*-expression s is* dynamically untyped *if it can be evaluated to a directly dynamically untyped expression using call-by-name evaluation, i.e.*

$$s \text{ is dynamically untyped}$$
$$\Longleftrightarrow$$
$$\exists t : s \xrightarrow{name,*} t \wedge t \text{ is directly dynamically untyped}$$

Note that dynamically untyped expressions diverge. The following proposition is true:

Proposition 4.1.12. *A closed* KFPT*-expression s is irreducible (w.r.t. call-by-name evaluation) iff one of the following conditions is true:*

- *s is WHNF or*
- *s is directly dynamically untyped.*

The proposition considers only closed expressions. Note also that not all divergent closed expressions are dynamically untyped, since $\Omega := (\lambda x.x\ x)\ (\lambda x.x\ x)$ diverges: $\Omega \xrightarrow{name} \Omega \xrightarrow{name} \Omega \ldots$.

In Haskell dynamically untyped expressions are discovered at *compile time*, because Haskell has a strong and static type system. A drawback is that there are KFPT-expressions that are not dynamically untyped, but are not typeable in Haskell (an example is the expression Ω, or the expression $\texttt{case}_{\texttt{Bool}}\ \texttt{True}\ \texttt{of}\ \{\texttt{True} \to \texttt{True}; \texttt{False} \to \texttt{Nil}\}$). We will discuss this again when we consider polymorphic typing of expressions.

4.1.4. Searching the Call-by-Name-Redex

As for the lambda calculus, we provide an *alternative* way to find the call-by-name redex by using a labeling algorithm. In compilers and interpreters of functional languages, expressions are represented as term graphs and they use a similar method, which is called *graph-unwinding*. For KFPT-expression s, the labeling algorithm starts with $s^\star$. The the following label shifting rules are applied exhaustively.

- $(s\ t)^\star \Rightarrow (s^\star\ t)$
- $(\texttt{case}_T\ s\ \texttt{of}\ Alts)^\star \Rightarrow (\texttt{case}_T\ s^\star\ \texttt{of}\ Alts)$

After executing the labeling algorithm, the following cases can occur:

- The label is at an abstraction. Then there are three subcases:
  - s is the labeled abstraction. Then an FWHNF is found and no call-by-name reduction is applicable.
  - The direct superterm of the abstraction is an application, i.e. s is of the form $C[(\lambda x.s')^\star\ t]$. Then the call-by-name reduction is $C[(\lambda x.s')\ t] \xrightarrow{name,\beta} C[s'[t/x]]$.
  - The direct superterm of the abstraction is a `case`-expression, i.e. s is of the form $C[\texttt{case}_T\ (\lambda x.s')^\star\ \texttt{of}\ Alts]$. Then s is directly dynamically untyped and no call-by-name reduction is applicable to s.
- The label is at an constructor application. Then there are the following subcases:
  - s is the labeled constructor application. Then a CWHNF is found and thus no call-by-name reduction is applicable
  - The direct superterm of the constructor application is a `case`-expression, i.e. s is of the form $C[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ Alts]$. There are two cases:
    - c belongs to type T. Then there is a matching `case`-alternative for c. Then reduce:

$$C[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ \{\ldots;(c\ x_1\ \ldots\ x_n) \rightarrow t\ ;\ldots\}] \xrightarrow{name,case} C[t[s_1/x_1,\ldots,s_n/x_n]]$$

    - c does not belong to type T. Then no call-by-name reduction is applicable to s and s is directly dynamically untyped.
  - The direct superterm is an application, i.e. s is of the form $C[(c\ s_1\ \ldots\ s_n)^\star\ t]$. Then no call-by-name reduction is applicable to s and s is directly dynamically untyped.
- The label is at a variable. Then s is of the form $C[x^\star]$. Then no call-by-name reduction is applicable to s, since a free variable was discovered (at reduction position). In this case s is not a WHNF, but also not directly dynamically untyped.

Example 4.1.13. *The call-by-name evaluation of the expression*

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \rightarrow \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \rightarrow (x\ z)\}\end{pmatrix}\texttt{True}))\ (\lambda u,v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

is as follows (where all labeling steps of the labeling algorithm are written explicitly);

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\texttt{True}))\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))^{\star}$$

$$\Rightarrow(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\texttt{True}))\ (\lambda u, v.v))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\Rightarrow(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\texttt{True}))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))^{\star}$$

$$\Rightarrow((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\texttt{True})^{\star}$$

$$\Rightarrow(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}^{\star}\ \texttt{True})$$

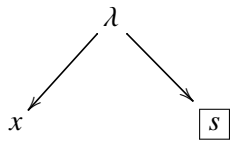
$$\Rightarrow(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\texttt{True})$$

$$\xrightarrow{name,case}(((\lambda u, v.v)\ (\lambda w.w))\ \texttt{True})^{\star}$$

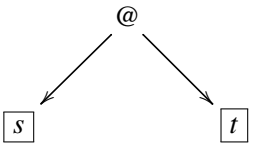
$$
\begin{aligned}
&\Rightarrow (((\lambda u, v.v)\ (\lambda w.w))^{\star}\ \texttt{True}) \\
&\Rightarrow (((\lambda u, v.v)^{\star}\ (\lambda w.w))\ \texttt{True}) \\
&\xrightarrow{name,\beta} ((\lambda v.v)\ \texttt{True})^{\star} \\
&\Rightarrow ((\lambda v.v)^{\star}\ \texttt{True}) \\
&\xrightarrow{name,\beta} \texttt{True}
\end{aligned}
$$

We explain the representation of KFPT-expressions as (syntax) trees: Every node of the tree represents a syntactic construct of the expression (starting with the root).

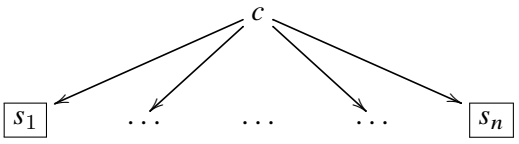
- variables are represented by nodes labeled with the variable.
- abstractions are represented by a node labeled with "λ" that has two children: the left child is the term representing the variable that is bound by the node and the right child is the term representing the body of the abstraction.



  where $\boxed{s}$ is the tree of s.
- applications are represented by a node labeled with "@" that has two children: one for the expression in function position and one for the argument. I.e., $(s\ t)$ is represented by

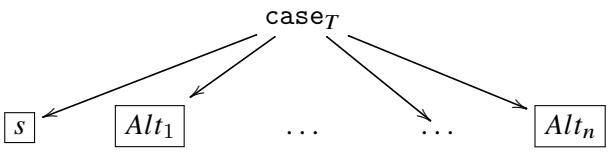


  where $\boxed{s}$ and $\boxed{t}$ are the trees for s and t.
- constructor applications have a node labeled with the constructor name c that has $\mathrm{ar}(c)$ children, one for each argument of the constructor application. I.e., $(c\ s_1 \ldots\ s_n)$ is represented by



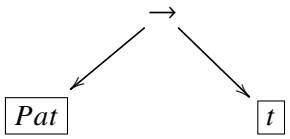
  where $\boxed{s_i}$ are the trees for s_i.
- `case`-expressions are represented by a node labeled with $\texttt{case}_T$ and having $n+1$ children, if n is the number of alternatives. The first child is for the first argument of the `case`-expression and the other children represent the alternatives. I.e.,

$\texttt{case}_T\ s\ \texttt{of}\ \{Alt_1; \ldots; Alt_n\}$ is represented by



where $\boxed{s}$ is the tree for s and $\boxed{Alt_i}$ is the tree of the i-th alternative.

- a `case`-alternative "pattern" $\rightarrow$ "expression" is represented by a node labeled with $\rightarrow$ that has two children: one for the pattern and one for the expression on the right-hand side. I.e., $Pat \rightarrow t$ is represented by

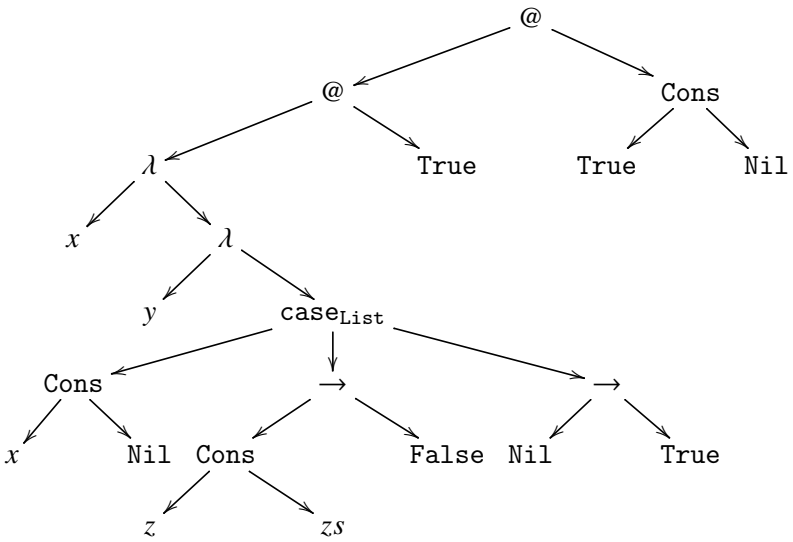


where $\boxed{Pat}$ is the tree of Pat and $\boxed{t}$ is the tree for t.

Example 4.1.14. *The tree for expression*

$$\left(\left(\begin{array}{l}\lambda x.\lambda y.\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ x\ \texttt{Nil})\ \texttt{of}\ \{\\ \quad (\texttt{Cons}\ z\ zs) \rightarrow \texttt{False};\\ \quad \texttt{Nil} \rightarrow \texttt{True}\}\end{array}\right)\ \texttt{True}\right)\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})$$

is:



Searching for the call-by-name redex can be summarized in the tree representation as follows: Walk along the left-most path until a node is found, that

- is labeled with a variable x, or
- is labeled with a constructor c, or
- is labeled with λ.

The direct superterm (or supertree) is the call-by-name redex, if a call-by-name reduction is applicable.

For the example tree, the search walks along the left-most path until it finds the first λ. The call-by-name-redex is the direct superterm, i.e. the application $(\lambda x.\ldots)$ `True`. We show the complete evaluation of the example:

$$\left(\left(\begin{array}{l}\lambda x.\lambda y.\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ x\ \texttt{Nil})\ \texttt{of}\ \{\\ \qquad (\texttt{Cons}\ z\ zs) \to \texttt{False};\\ \qquad \texttt{Nil} \to \texttt{True}\}\end{array}\right)\ \texttt{True}\right)\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})$$

$$\xrightarrow{name,\beta} \quad \left(\begin{array}{l}\lambda y.\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})\ \texttt{of}\ \{\\ \qquad (\texttt{Cons}\ z\ zs) \to \texttt{False};\\ \qquad \texttt{Nil} \to \texttt{True}\}\end{array}\right)\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})$$

$$\xrightarrow{name,\beta} \quad \texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})\ \texttt{of}\ \{(\texttt{Cons}\ z\ zs) \to \texttt{False}; \texttt{Nil} \to \texttt{True}\}$$

$$\xrightarrow{name,case} \quad \texttt{False}$$

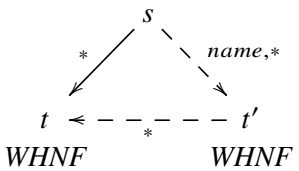
4.1.5. Properties of the Call-by-Name Reduction

For the call-by-name reduction the following can be proved by checking the possible cases:

- The call-by-name reduction is deterministic, i.e. for every KFPT-expression s, there is at most one expression t with $s \xrightarrow{name} t$.
- A WHNF is irreducible w.r.t. call-by-name evaluation.

As in the lambda calculus, call-by-name evaluation is standardising (the proof is similar to the proof in the lambda calculus, but would require to adapt the notion of parallel reduction to the new syntax and including (case)-reduction, we omit it).

Theorem 4.1.15. *Let s be a* KFPT*-expression. If $s \xrightarrow{*} t$ with (β)- and $(case)$-reductions (applied in arbitrary contexts), where t is a WHNF, then there exists a WHNF t', such that $s \xrightarrow{name,*} t'$ and $t' \xrightarrow{*} t$ (modulo α-equivalence). I.e., the following diagram illustrates the claim where straight lines are given reductions and dashed lines are existentially quantified reductions:*



A consequence of the theorem is, that (β)- and (case)-steps (applied at arbitrary positions) do not change the convergence of the expressions (in fact, they are correct program transformations w.r.t. the obvious definition of contextual equivalence for KFPT).

4.2. The Core Language KFPTS

We now extend KFPT to KFPTS. The "S" means supercombinators, which are names (or constants) that denote (recursive) functions.

4.2.1. Syntax

We assume that there is a set of supercombinator names $\mathcal{SC}$.

Definition 4.2.1. *Expressions of the language* KFPTS *are built by the following grammar where the side conditions on* `case`*-expressions must hold as in* KFPT*:*

$$
\begin{array}{lcl}
\mathbf{Expr} & ::= & V \mid \lambda V.\mathbf{Expr} \mid (\mathbf{Expr}_1\ \mathbf{Expr}_2) \\
& \mid & (c_i\ \mathbf{Expr}_1\ \ldots\ \mathbf{Expr}_{\mathrm{ar}(c_i)}) \\
& \mid & (\mathtt{case}_T\ \mathbf{Expr}\ \mathtt{of}\ \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_n \to \mathbf{Expr}_n\}) \\
& \mid & SC \text{ where } SC \in \mathcal{SC} \\
\\
\mathbf{Pat}_i & ::= & (c_i\ V_1 \ldots V_{ar(c_i)}) \text{ where variables } V_i \text{ are pairwise distinct.}
\end{array}
$$

For every supercombinator, there must exist exactly one *supercombinator definition*:

Definition 4.2.2. *A* supercombinator definition *is an equation*

$$SC\ V_1\ \ldots\ V_n = \mathbf{Expr}$$

where V_i are pairwise distinct variables and **Expr** *is a* KFPTS*-expression such that $FV(\mathbf{Expr}) \subseteq \{V_1, \ldots, V_n\}$, i.e. only $V_1, \ldots, V_n$ can occur free in* **Expr**. *We write* $\mathrm{ar}(SC) = n$ *to denote the* arity *of the supercombinator SC.*

We assume that names of supercombinators, of variables, and of constructors do not overlap (all sets are disjoint).

Definition 4.2.3. *A* KFPTS-program *consists of*

- *a set of types and data constructors,*
- *a set of supercombinator definitions,*
- *and a* KFPTS*-expression s (we could also define it as a special supercombinator named* `main` *with definition* `main` = *s.).*

As a side condition it is required that the supercombinators that occur in the right-hand sides of the definitions and in s are all defined.

4.2.2. Call-by-Name Evaluation of KFPTS-Expressions

The KFPT-Call-by-Name evaluation has to be extended, to evaluate supercombinator applications. Reduction contexts in KFPTS are defined analogously to KFPT as:

$$\textbf{RCtxt} ::= [\cdot] \mid (\textbf{RCtxt Expr}) \mid \texttt{case}_T\ \textbf{RCtxt}\ \texttt{of}\ Alts$$

The reduction rules are extended by one rule:

Definition 4.2.4. *The reduction rules* (β)*,* $(case)$ *and* $(SC\text{-}\beta)$ *are defined in* KFPTS *as:*

$$\begin{array}{ll}
(\beta) & (\lambda x.s)\ t \xrightarrow{\beta} s[t/x] \\
(case) & \texttt{case}_T\ (c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ \texttt{of}\ \{\ldots;\ (c\ x_1 \ldots\ x_{\mathrm{ar}(c)}) \to t;\ \ldots\} \\
 & \qquad \xrightarrow{case} t[s_1/x_1, \ldots, s_{\mathrm{ar}(c)}/x_{\mathrm{ar}(c)}] \\
(SC\text{-}\beta) & (SC\ s_1\ \ldots\ s_n) \xrightarrow{SC\text{-}\beta} e[s_1/x_1, \ldots, s_n/x_n], \\
 & \quad \textit{if } SC\ x_1\ \ldots\ x_n = e \textit{ is the definition of } SC
\end{array}$$

The call-by-name reduction in KFPTS applies one of the three rules:

Definition 4.2.5. *If* $r_1 \to r_2$ *with a* (β)*-,* $(case)$*- or* $(SC\text{-}\beta)$*-reduction, then for any reduction context* R*,* $R[r_1] \xrightarrow{name} R[r_2]$ *is a (*KFPTS*)-call-by-name reduction.*

The definition of WHNFs has to be slightly adapted, since we allow occurrences of supercombinators that are not fully saturated, i.e. expressions of the form $SC\ s_1\ \ldots\ s_m$ where $\mathrm{ar}(SC) > m$. They behave like abstractions $\lambda x_{m+1}, \ldots, x_{\mathrm{ar}(SC)}.SC\ s_1\ \ldots\ s_m\ x_{m+1} \ldots\ x_{\mathrm{ar}(SC)}$ and thus they belong to the FWHNFs in KFPTS. The definition of CWHNFs is the same as in KFPT.

For the dynamic typing rules, the following case is added

- $R[\texttt{case}_T\ SC\ s_1\ \ldots\ s_m\ \texttt{of}\ Alts]$ is directly dynamically untyped if $\mathrm{ar}(SC) > m$.

The labeling algorithm for searching the redex, uses the same shifting rules as in KFPT, but after labeling, there are new cases:

- The labeled subexpression is a supercombinator, i.e. the labeled expression is of the form $C[SC^\star]$. Then there are three subcases:
  - $C = C'[[\cdot]\ s_1\ \ldots\ s_n]$ and $\mathrm{ar}(SC) = n$. Then apply the (SC-β)-reduction: $C'[SC\ s_1\ \ldots\ s_n] \xrightarrow{name,SC\text{-}\beta} C'[e[s_1/x_1, \ldots, s_n/x_n]]$ (if $SC\ x_1\ \ldots\ x_n = e$ is the definition of SC).
  - $C = [[\cdot]\ s_1\ \ldots\ s_m]$ und $\mathrm{ar}(SC) > m$. Then the expression is an FWHNF.
  - $C = C'[\texttt{case}_T\ [\cdot]\ s_1\ \ldots\ s_m\ \texttt{of}\ Alts]$ and $\mathrm{ar}(SC) > m$. Then the expression is directly dynamically untyped.

Example 4.2.6. *Assume that the supercombinators* map *and* not *are defined as:*

$$\begin{array}{l}
map\ f\ xs = \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys) \to \texttt{Cons}\ (f\ y)\ (map\ f\ ys)\} \\
not\ x \quad\ = \texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to \texttt{False}; \texttt{False} \to \texttt{True}\}
\end{array}$$

Call-by-name evaluation of expression $map\ not$ (`Cons True (Cons False Nil)`) *is:*

$$\begin{array}{ll} & map\ not\ (\texttt{Cons True (Cons False Nil)}) \\ \xrightarrow{name,SC\text{-}\beta} & \texttt{case}_{\texttt{List}}\ (\texttt{Cons True (Cons False Nil)})\ \texttt{of}\ \{ \\ & \quad \texttt{Nil} \rightarrow \texttt{Nil}; \\ & \quad (\texttt{Cons}\ y\ ys) \rightarrow \texttt{Cons}\ (not\ y)\ (map\ not\ ys)\} \\ \xrightarrow{name,case} & \texttt{Cons}\ (not\ \texttt{True})\ (map\ not\ (\texttt{Cons False Nil})) \end{array}$$

The obtained expression is a WHNF. To evaluate the full list, one has to wrap the expression into a context that forces the evaluation.

4.3. Extension by `seq`

Haskell has the binary operator `seq` which has the following semantics:

$$(\texttt{seq}\ a\ b) = \begin{cases} b & \text{if } a{\downarrow} \\ \bot & \text{if } a{\Uparrow} \end{cases}$$

Operationally, for the evaluation of an expression (`seq` $a\ b$), first a is evaluated to a WHNF v, and after obtaining `seq` $v\ b$ the reduction is (`seq` $v\ b$) $\rightarrow b$. Using `seq`, one can define the operator `$!` as

$$f\ \texttt{\$!}\ x = \texttt{seq}\ x\ (f\ x)$$

which makes a function strict in its argument: Let $f\ x = t$ be a supercombinator definition, then f `$!` s only returns $f[s/x]$ if $s{\downarrow}$, otherwise $(f\ s){\Uparrow}$. Operationally, this can be implemented by evaluating first s and thereafter performing an $\xrightarrow{SC\text{-}\beta}$-step when the argument is a value (i.e. it mimics call-by-value instead of call-by-name evaluation for f).

Note that the implementation of `$!` using `seq` makes more sense, if sharing (or call-by-need) is used in the evaluation, since it the expression replaced for x should be evaluated once and not twice. Semantically, it makes no difference, but for efficiency it makes sense.

The operators `seq` and `$!` are helpful, to enforce strict evaluation which is sometimes advantageous to optimize the space-behavior during evaluation. Let us consider a function to compute the factorial of a natural number (we assume an implementation of numbers and operations on it). A naive implementation is

$$fac\ x = \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * (fac\ (x-1))$$

The call-by-name evaluation of `fac` n will generate an expression of the form $n*(n-1)*\ldots*1$, and thereafter computing all the multiplications. Thus, representing this intermediate expression

will require space that is linear in n. A first approach is to use an end-recursive variant:

$$\begin{aligned} fac\ x &= facER\ x\ 1 \\ facER\ x\ y &= \texttt{if}\ x = 0\ \texttt{then}\ y\ \texttt{else}\ facER\ (x-1)\ (x*y) \end{aligned}$$

However, the space problem is not solved, since the second argument of $facER$ still requires linear space in n. Using sharing and `seq`, the following definition in Haskell only requires constant space:

```
fac x = facER x 1
 where facER 0 y = y
       facER x y = let x' = x-1
                       y' = x*y
                   in seq x' (seq y' (facER x' y'))
```

The languages KFPT and KFPTS do not have `seq` and it cannot be encoded in them. Thus, we introduce KFPT+seq and KFPTS+seq as the calculi that extend KFPT or KFPTS with the operator `seq`. We leave the definition of the extended syntax and reduction contexts as an exercise. The new required reduction rule is

$$\texttt{seq}\ v\ t \to t, \text{ if } v \text{ is a WHNF}$$

4.4. Polymorphic Types

All considered languages are untyped or very weakly typed. Haskell uses a strong static and polymorphic type system. With KFPTSP (KFPTSP+seq, resp.) we denote the core language KFPTS (KFPTS+seq, resp.) where the set of expressions and programs are restricted to expressions and programs that are *well-typed*. We will investigate the polymorphic typing in the next chapter, but we already introduce the syntax for types:

Definition 4.4.1. *The syntax of polymorphic types is given by the following grammar:*

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1\ \ldots\ \mathbf{T}_n \mid \mathbf{T}_1 \to \mathbf{T}_2$$

where TV is a non-terminal generating a type variable and TC is a type constructor of arity n.

Type constructors are names like `Bool`, `List`, `Pair`, etc. Type constructors may require types as arguments, for instance, the type constructor `List` requires an argument type, which fixes the type of the elements of the list, e.g. `List Bool` is the type of a list of booleans. We sometimes use Haskell-notation and write `[Bool]` instead of `List Bool`. The number of required arguments is the arity of the type constructor, sometimes written as $\mathrm{ar}(TC)$.

Type type $T_1 \to T_2$ is a function type, for a function that receives an argument of type T_1 and returns a result of type T_2. For instance, the test whether a number is even has type `Int` $\to$

`Bool`. The $\to$ in types is right-associative, i.e. $T_1 \to T_2 \to T_3$ means $T_1 \to (T_2 \to T_3)$ and *not* $(T_1 \to T_2) \to T_3$ (which is *different* type).

Type-constructors of arity 0, are called *base types*. Types that do not contain type variables are called *monomorphic types*, and types that may contain type variables are called *polymorphic types*. If a type variable occurs in a type, then this type represents a set of monomorphic types, since the variable may be substituted by any type. For example, the identity function has type $a \to a$ where a is a type variable. It represents all types where a is replaced a type (such a replacement is called a type substitution). For example, the substitution $\sigma = \{a \mapsto \texttt{List Bool}\}$ applied to $a \to a$ results in $\sigma(a \to a) = \texttt{List Bool} \to \texttt{List Bool}$

In Haskell, one writes $e :: T$ if expression e is of type T. We also use this notation.

Some examples are:

$$\begin{array}{lcl} \texttt{True} & :: & \texttt{Bool} \\ \texttt{False} & :: & \texttt{Bool} \\ \texttt{not} & :: & \texttt{Bool} \to \texttt{Bool} \\ \texttt{map} & :: & (a \to b) \to [a] \to [b] \\ (\lambda x.x) & :: & (a \to a) \end{array}$$

We will discuss type checking and type inference in detail in the next chapter, but introduce some *simplified* typing rules now, to get a first impression. The notation of the rules is

$$\frac{\text{premises}}{\text{conclusion}},$$

i.e. to derive the conclusion, the premises have to be satisfied. The simplified rules are:

- For the application:
$$\frac{s :: T_1 \to T_2, t :: T_1}{(s\ t) :: T_2}$$
- Instantiation:
$$\frac{s :: T}{s :: T'} \quad \text{if } T' = \sigma(T) \text{ for type substitution } \sigma, \text{ that replaces type variables with types.}$$
- For `case`-expressions:
$$\frac{s :: T_1, \quad \forall i : Pat_i :: T_1, \quad \forall i : t_i :: T_2}{(\texttt{case}_T\ s\ \texttt{of}\ \{Pat_1 \to t_1; \ldots; Pat_n \to t_n\}) :: T_2}$$

Example 4.4.2. *The boolean connectives* `and` *and* `or` *can be defined in* KFPTS *as:*

$$\begin{array}{ll} \texttt{and} & := \lambda x, y.\texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to y; \texttt{False} \to \texttt{False}\} \\ \texttt{or} & := \lambda x.y.\texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to \texttt{True}; \texttt{False} \to y\} \end{array}$$

The expression and True False *can be typed by applying the rule for the application twice:*

$$\dfrac{\dfrac{\texttt{and} :: \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}, \texttt{True} :: \texttt{Bool}}{(\texttt{and}\ \texttt{True}) :: \texttt{Bool} \to \texttt{Bool}}, \texttt{False} :: \texttt{Bool}}{(\texttt{and}\ \texttt{True}\ \texttt{False}) :: \texttt{Bool}}$$

Example 4.4.3. *The expression*

$$\texttt{case}_{\texttt{Bool}}\ \texttt{True}\ \texttt{of}\ \{\texttt{True} \to (\texttt{Cons}\ \texttt{True}\ \texttt{Nil}); \texttt{False} \to \texttt{Nil}\}$$

can be typed as follows:

$$\dfrac{\texttt{True} :: \texttt{Bool}, \texttt{False} :: \texttt{Bool}, \dfrac{\dfrac{\dfrac{\texttt{Cons} :: a \to [a] \to [a]}{\texttt{Cons} :: \texttt{Bool} \to [\texttt{Bool}] \to [\texttt{Bool}]}, \texttt{True} :: \texttt{Bool}}{(\texttt{Cons}\ \texttt{True}) :: [\texttt{Bool}] \to [\texttt{Bool}]}, \dfrac{\texttt{Nil} :: [a]}{\texttt{Nil} :: [\texttt{Bool}]}}{(\texttt{Cons}\ \texttt{True}\ \texttt{Nil}) :: [\texttt{Bool}]}, \dfrac{\texttt{Nil} :: [a]}{\texttt{Nil} :: [\texttt{Bool}]}}{\texttt{case}_{\texttt{Bool}}\ \texttt{True}\ \texttt{of}\ \{\texttt{True} \to (\texttt{Cons}\ \texttt{True}\ \texttt{Nil}); \texttt{False} \to \texttt{Nil}\} :: [\texttt{Bool}]}$$

Example 4.4.4. *The operator* seq *is of type $a \to b \to b$, since it ignores the first argument in the result.*

Example 4.4.5. *If types of* map *and* not *are already given, then (*map not*) can be typed as follows:*

$$\dfrac{\dfrac{\texttt{map} :: (a \to b) \to [a] \to [b]}{\texttt{map} :: (\texttt{Bool} \to \texttt{Bool}) \to [\texttt{Bool}] \to [\texttt{Bool}]}, \texttt{not} :: \texttt{Bool} \to \texttt{Bool}}{(\texttt{map}\ \texttt{not}) :: [\texttt{Bool}] \to [\texttt{Bool}]}$$

Note that it is not deterministic when and how to apply the instantiation rule, here we did it by guessing the right position and the substitution. Later we will provide an algorithm which removes this guessing.

We did not provide a typing rule for abstractions and the rule for case is not precise, since it ignores the correspondence between the types of the variables in patterns and right-hand sides of alternatives. Additionally, we are not able to type recursive supercombinators, since there is no rule. All this will be done in the next chapter.

4.5. Conclusion and References

We summarize the introduced core languages (which extend the call-by-name lambda calculus) in a table:

Core Language	Description
KFPT	Extension of the call-by-name lambda calculus with weakly typed `case` and data constructors `seq` is not encodable.
KFPTS	Extension of KFPT by recursive supercombinators
KFPTSP	Restriction of KFPTS to well-typed expressions using a polymorphic type system
KFPT+seq	Extension of KFPT with the `seq`-operator
KFPTS+seq	Extension of KFPTS with the `seq`-operator
KFPTSP+seq	Restriction of KFPTS+seq to well-typed expressions using a polymorphic type system

The core languages can be found in (Schmidt-Schauß, 2009), similar core languages can for instance be found in (Peyton Jones, 1987).

We view the language KFPTSP+seq as a core language of Haskell.

5. Polymorphic Type Inference

In this chapter we will introduce two algorithms for polymorphic type inference for recursive functional programs, i.e. we will consider the core language KFPTSP+seq. Before introducing the algorithms, we motivate why a type system should be used, in particular, we discuss the advantages over dynamic typing. We then explain the task of unification for types which is necessary for the type inference algorithms. We then introduce typing of KFPTS+seq-expressions and at the end we consider the typing of (recursive) supercombinators, where we consider *iterative typing* and *Hindley-Damas-Milner typing*. While iterative typing computes most general polymorphic types, the decision problem of whether a program is iteratively typeable is undecidable. In contrast, Hindley-Damas-Milner-typing is decidable, but it computes less general types.

5.1. Motivation

Since KFPTS+seq-programs are not typed, dynamic type errors can occur at runtime. Such type errors are programming errors, i.e. a programmer does not assume that a program has type errors. A strong and static type system prevents from such type errors and thus helps the programmer to detect errors at compile time when type checking is performed.

Types can also be used to document the program. Often the type of a function gives a good idea of what the function does or does not do. Therefore, a good type system should meet the following requirements:

- Type checking should be performed at compile time.
- Typed programs do not lead to type errors at runtime.

Other desirable properties for a strong static type system are

- The type system should not be such restrictive that well-typed programs are difficult to write, i.e. useful programs should not be rejected because of the type system.
- Ideally, the type system should not require the programmer to write down all the types, i.e. instead of just checking the types, the type system should ideally be able to compute the type itself (this is called *type inference*). If type inference is supported, the system should compute general types (ideally the most general types).

There are type systems that do not satisfy all the desired properties. For example, the *simply typed* lambda calculus is very restrictive: typing is decidable at compile time, but the set of typed programs is no longer Turing complete, since all simply typed programs terminate (this can be proved!). In extensions of Haskell's type system, type inference is no longer possible, i.e. there

are cases where types must be provided by the programmer. There are also extensions where type checking is undecidable, which can lead to the problem that the compiler may not terminate during type check.

A first approach to a general type system for KFPTSP+seq would be to set:

> A KFPTSP+seq-program is well-typed, if it cannot lead to a dynamic type error during runtime.

This restricts the set of well-typed programs, to those that are not directly dynamically untyped. We sketch the proof. Let `tmEncode` be a KFPTS+seq-supercombinator that behaves like a universal Turing machine (see Chapter A for a Haskell-function `tmEncode`), i.e. it receives an encoding of a Turing machine and an input and simulates the TM on the input. It returns `True`, if the TM halts on the input. We assume that `tmEncode` is not dynamically untyped (this holds for the Haskell-program in the Chapter A, since it is Haskell-typeable, which implies that it is not dynamically untyped).

For TM encoding *enc* and an input *inp*, let the expression *s* be defined as

$$
\begin{array}{rl}
s := & \texttt{if tmEncode } enc\ inp \\
& \quad \texttt{then case}_{\text{Bool}}\ \texttt{Nil of } \{\texttt{True} \rightarrow \texttt{True}; \texttt{False} \rightarrow \texttt{False}\} \\
& \quad \texttt{else case}_{\text{Bool}}\ \texttt{Nil of } \{\texttt{True} \rightarrow \texttt{True}; \texttt{False} \rightarrow \texttt{False}\}
\end{array}
$$

Since (`tmEncode` *enc inp*) is not dynamically untyped, the following holds: *s* is dynamically untyped if, and only if, the evaluation of `tmEncode` *enc inp*) ends with `True`. So: if we could decide whether *s* is dynamically untyped, we could decide the halting problem, which is impossible. This shows:

Proposition 5.1.1. *The dynamic typing of* KFPTS+seq*-programs is undecidable.*

Hence, we consider type systems that restrict the programs and the programming. For both type systems that we consider, the following will hold:

- A well-typed expression is not dynamically untyped (otherwise the type system would not be of much help).
- There are expressions, that are not dynamically untyped, but are also not well-typed.

5.2. Types: Definitions, Notation and Unification

In this section we introduce definitions and notations for types. We recall the syntax of polymorphic types:

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1\ \ldots\ \mathbf{T}_n \mid \mathbf{T}_1 \rightarrow \mathbf{T}_2$$

where TV is a type variable and TC is a type constructor[1].

[1] We remind that the list type is represented in Haskell as `[a]`. This more or less means that the type constructor is `[]` applied to the variable a. A special syntax is also used for tuple types: `(,...,)` is the type constructor, e.g. `(a,b)` is the type for pairs. We will use this syntax in the following.

As a reminder: A *base type* is a type of the form TC, where TC is of arity 0. A monomorphic type) is a type that does not contain any type variables.

Example 5.2.1. *The types* `Int`, `Bool` *and* `Char` *are base types. The types* `[Int]` *and* `Char -> Int` *are not base types, but monomorphic types. The types* `[a]` *and* `a -> a` *are neither base types nor monomorphic types.*

We write $Vars(T)$ for the set of variables that occur in the polymorphic type T.

We also use *universally quantified types* as a new notation. Let τ be a polymorphic type with occurrences of the type variables $\alpha_1, \ldots, \alpha_n$, then $\forall\alpha_1, \ldots, \alpha_n.\tau$ is the *universally quantified type* for τ. Polymorphic types can be considered as universally quantified, since the type variables represent any type. We use the quantifier syntax to distinguish (in the typing procedure) between types that can be 'copied' (i.e. renamed), and types that we cannot rename in the typing procedure. The order of the quantified type variables in the quantifier is irrelevant. Therefore, we also use the notation $\forall\mathcal{X}.\tau$, where $\mathcal{X}$ is a set of type variables.

Definition 5.2.2. *A* type substitution *is a mapping $\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ of a finite set of type variables to types. Let σ be a type substitution. The homomorphic extension σ_E of σ is the extension of type substitution as a mapping from types to types, which is defined by:*

$$\begin{array}{rcl}
\sigma_E(TV) & := & \sigma(TV), \text{ if } \sigma \text{ maps the variable } TV \\
\sigma_E(TV) & := & TV, \text{ if } \sigma \text{ does not represent the variable } TV \\
\sigma_E(TC\ T_1\ \ldots\ T_n) & := & TC\ \sigma_E(T_1)\ \ldots\ \sigma_E(T_n) \\
\sigma_E(T_1 \to T_2) & := & \sigma_E(T_1) \to \sigma_E(T_2)
\end{array}$$

In the following, we do not distinguish between σ and the extension σ_E.

A type substitution is a ground substitution for a type τ *if and only if σ maps to monomorphic types and all type variables that occur in τ are mapped to monomorphic types by σ.*

Using ground substitutions, we can define a semantics of polymorphic types that uses monomorphic types: Let τ be a polymorphic type, the ground type semantics $\mathrm{sem}(\tau)$ of τ is the of all monomorphic types, that can be generated from τ by applying ground substitutions, i.e.

$$\mathtt{sem}(\tau) := \{\sigma(\tau) \mid \sigma \text{ is a ground substitution for } \tau\}$$

This corresponds to the intuition of type schemas: a polymorphic type describes the schema of a set of monomorphic types.

We introduced some simple typing rules in Section 4.4. Let us reconsider the rule for the application:

$$\frac{s :: T_1 \to T_2, t :: T_1}{(s\ t) :: T_2}$$

This rule requires that the argument type of s already matches the type of t. As an example, let us assume that we know the most general types of `map` and `not` as:

```
map :: (a -> b) -> [a] -> [b]
not :: Bool -> Bool
```

To type `map not` using the application rule, we have to instantiate the type of `map`. The required substitution is $\sigma = \{\texttt{a} \mapsto \texttt{Bool}, \texttt{b} \mapsto \texttt{Bool}\}$, however, we do not want to guess the right substitution, we want to compute it. A (general) procedure to "search for a substitution that makes types (or also terms) equal" is *unification*. We define this now:

Definition 5.2.3. *A* unification problem *on types is a set E of equations of the form $\tau_1 \doteq \tau_2$ where τ_1 and τ_2 are polymorphic types. A solution to a unification problem on types is a substitution σ (called* unifier*), such that $\sigma(\tau_1) = \sigma(\tau_2)$ for all equations $\tau_1 \doteq \tau_2$ of E.*

A most general solution *(most general unifier, mgu) of E is a unifier σ such that for every unifier ρ of E there is a substitution γ such that $\rho(x) = \gamma \circ \sigma(x)$ for all $x \in Vars(E)$.*

One can verify that most general unifiers are unique up to the renaming of variables.

Example 5.2.4. *For the unification problem $\{\alpha \doteq \beta\}$, the substitutions $\sigma = \{\alpha \mapsto \beta\}$ and $\sigma' = \{\beta \mapsto \alpha\}$ are most general unifiers. The substitution $\sigma'' = \{\alpha \mapsto \texttt{Bool}, \beta \mapsto \texttt{Bool}\}$ is a unifier, but not a most general unifier.*

?? 1 computes a most general unifier, if it exists. The data structure it uses, is a multi-set of equations to be unified (multi-sets are analogous to sets, but multiple occurrences of elements are allowed). We use E for such multisets and $E \cup E'$ means the disjoint union of two multisets. The notation $E[\tau/\alpha]$ means that in all equations of E the type variable α is replaced by type τ (i.e. on all left and right-hand sides of the equations the substitution $\{\alpha \mapsto \tau\}$ is applied).

The unification algorithm has the following properties (for a proof see e.g. (Baader & Nipkow, 1998)):

- The algorithm stops with Fail iff the input has no solution.
- The algorithm stops with success iff the input has a unifier. The equation system E then has the form $\{\alpha_1 \doteq \tau_1, \ldots, \alpha_n \doteq \tau_n\}$, where α_i are pairwise distinct type variables, each α_i does not occur in any τ_j. The unifier is $\sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$.
- If the algorithm computes a unifier, then it is a most general unifier.
- Rules can be applied in any order, no backtracking is necessary, i.e. the algorithm can be implemented in a deterministic way.
- The algorithm terminates for every input.
- Types in the resulting substitution can grow exponentially with the size of the input
- The unification algorithm can be implemented with time complexity $O(n \log n)$. This requires to use sharing (to avoid the exponentially large types) and a different solve rule.
- The unification problem (i.e. the question whether a set of type equations is unifiable) is P-complete, i.e. roughly speaking all PTIME-problems can be represented as unification problem. As a consequence, unification is not efficiently parallelizable.

Algorithm 1: Unification Algorithm for Types

The unification algorithm starts with the unification problem and then applies the following rules exhaustively, where no order of the rule application is necessary, i.e. rule application is non-deterministic. It stops if no rule can be applied, or Fail occurs.

$$\text{(Fail1)}\ \frac{E \cup \{(TC_1\ \tau_1\ \ldots\ \tau_n) \doteq (TC_2\ \tau'_1\ \ldots\ \tau'_m)\}}{\text{Fail}}\ \text{if } TC_1 \neq TC_2$$

$$\text{(Fail2)}\ \frac{E \cup \{(TC_1\ \tau_1\ \ldots\ \tau_n) \doteq (\tau'_1 \to \tau'_2)\}}{\text{Fail}} \qquad \text{(Fail3)}\ \frac{E \cup \{(\tau'_1 \to \tau'_2) \doteq (TC_1\ \tau_1\ \ldots\ \tau_n)\}}{\text{Fail}}$$

$$\text{(Decompose1)}\ \frac{E \cup \{TC\ \tau_1\ \ldots\ \tau_n \doteq TC\ \tau'_1\ \ldots\ \tau'_n\}}{E \cup \{\tau_1 \doteq \tau'_1, \ldots, \tau_n \doteq \tau'_n\}} \qquad \text{(Decompose2)}\ \frac{E \cup \{\tau_1 \to \tau_2 \doteq \tau'_1 \to \tau'_2\}}{E \cup \{\tau_1 \doteq \tau'_1, \tau_2 \doteq \tau'_2\}}$$

$$\text{(Orient)}\ \frac{E \cup \{\tau_1 \doteq \alpha\}}{E \cup \{\alpha \doteq \tau_1\}}\ \text{if } \tau_1 \text{ is not a type variable and } \alpha \text{ is a type variable} \qquad \text{(Elim)}\ \frac{E \cup \{\alpha \doteq \alpha\}}{E}\ \text{where } \alpha \text{ is a type variable}$$

$$\text{(Solve)}\ \frac{E \cup \{\alpha \doteq \tau\}}{E[\tau/\alpha] \cup \{\alpha \doteq \tau\}}\ \text{if type variable } \alpha \text{ does not occur in } \tau, \text{ but occurs in } E \qquad \text{(OccursCheck)}\ \frac{E \cup \{\alpha \doteq \tau\}}{\text{Fail}}\ \text{if } \tau \neq \alpha \text{ and type variable } \alpha \text{ occurs in } \tau$$

We sketch the termination proof: Let E be a unification problem and

- $Var(E)$ be the number of *unsolved* type variables in E, where a variable α is solved iff it occurs once in E as the left hand side of an equation (i.e. $E = E' \cup \{\alpha \doteq \tau\}$ where $\alpha \notin Vars(E') \cup Vars(\tau)$).
- $Size(E)$ is the sum of all sizes of the types on right-hand and left sides of equations in E, where the size of a type is `tsize` defined as: $\texttt{tsize}(TV) = 1$, $\texttt{tsize}(TC\ T_1\ \ldots\ T_n) = 1 + \sum_{i=1}^{n} \texttt{tsize}(T_i)$ and $\texttt{tsize}(T_1 \to T_2) = 1 + \texttt{tsize}(T_1) + \texttt{tsize}(T_2)$
- $OEq(E)$ is the number of not oriented equations in E, where an equation is oriented, if it is of the form $\alpha \doteq \tau$ where α is a type variable.
- $M(E) = (\,Var(E), Size(E), OEq(E))$, where $M(\text{Fail}) := (-1, -1, -1)$.

For the termination proof, we show that for each rule $\frac{E}{E'}$ we have $M(E') <_{lex} M(E)$, where $<_{lex}$ is the lexicographic order on triples. This is sufficient since the measure M is well-founded. The Fail-rules and rule (OccursCheck) always strictly decrease the measure. The Decompose-rules do not increase $Var(.)$, but strictly decrease $Size(.)$ (and may increase $OEq(.)$). The rule (Orient) does not increase $Var(.)$, does not change $Size(.)$, but strictly decreases $OEq(.)$. The (Elim)-rule does not increase $Var(.)$, but strictly decreases $Size(.)$. The (Solve)-rule strictly decreases $Var(.)$ (and may increase $Size(.)$).

Example 5.2.5. *The unification problem* $\{(a \to b) \doteq \texttt{Bool} \to \texttt{Bool}\}$ *is solved by the unification algorithm in one step:*

$$\textit{(Decompose2)}\ \frac{\{(a \to b) \doteq \texttt{Bool} \to \texttt{Bool}\}}{\{a \doteq \texttt{Bool}, b \doteq \texttt{Bool}\}}$$

Example 5.2.6. *Consider the unification problem* $\{a \to [a] \doteq \texttt{Bool} \to c, [d] \doteq c\}$. *An execution of unification algorithm is:*

$$\begin{array}{r} \textit{(Decompose2)}\ \dfrac{\{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\}}{\{[d] \doteq c, a \doteq \texttt{Bool}, [a] \doteq c\}} \\ \textit{(Orient)}\ \dfrac{\{[d] \doteq c, a \doteq \texttt{Bool}, [a] \doteq c\}}{\{[d] \doteq c, a \doteq \texttt{Bool}, c \doteq [a]\}} \\ \textit{(Solve)}\ \dfrac{\{[d] \doteq c, a \doteq \texttt{Bool}, c \doteq [a]\}}{\{[d] \doteq [a], a \doteq \texttt{Bool}, c \doteq [a]\}} \\ \textit{(Solve)}\ \dfrac{\{[d] \doteq [a], a \doteq \texttt{Bool}, c \doteq [a]\}}{\{[d] \doteq [\texttt{Bool}], a \doteq \texttt{Bool}, c \doteq [\texttt{Bool}]\}} \\ \textit{(Decompose1)}\ \dfrac{\{[d] \doteq [\texttt{Bool}], a \doteq \texttt{Bool}, c \doteq [\texttt{Bool}]\}}{\{d \doteq \texttt{Bool}, a \doteq \texttt{Bool}, c \doteq [\texttt{Bool}]\}} \end{array}$$

The unifier is $\{d \mapsto \texttt{Bool}, a \mapsto \texttt{Bool}, c \mapsto [\texttt{Bool}]\}$.

Example 5.2.7. *The unification problem* $\{a \doteq [b], b \doteq [a]\}$ *has no solution:*

$$\begin{array}{r} \textit{(Solve)}\ \dfrac{\{a \doteq [b], b \doteq [a]\}}{\{a \doteq [[a]], b \doteq [a]\}} \\ \textit{(OccursCheck)}\ \dfrac{\{a \doteq [[a]], b \doteq [a]\}}{\textit{Fail}} \end{array}$$

The unification problem $\{a \to [b] \doteq a \to c \to d\}$ *also has no solution:*

$$\begin{array}{r} \textit{(Decompose2)}\ \dfrac{\{a \to [b] \doteq a \to c \to d\}}{\{a \doteq a, [b] \doteq c \to d\}} \\ \textit{(Elim)}\ \dfrac{\{a \doteq a, [b] \doteq c \to d\}}{\{[b] \doteq c \to d\}} \\ \textit{(Fail2)}\ \dfrac{\{[b] \doteq c \to d\}}{\textit{Fail}} \end{array}$$

Exercise 5.2.8. *Solve the unification problems:*

- $\{\texttt{BTree}\ a \doteq \texttt{BTree}\ (b \to c), [b] \doteq [[c]], d \to e \to f = \texttt{Bool} \to c\}$
- $\{a \to g \to (b \to c) \to (d \to e) \doteq a \to f, ([b] \to [c] \to [d] \to ([e] \to [g])) \to h \doteq f \to h\}$

We will now often use unification without providing the calculation of the unifier.

5.3. Polymorphic Typing of KFPTSP+seq-Expressions

First we will consider the typing of expressions, later we will consider supercombinators. For now we will assume that the supercombinators are already typed.

Since we have introduced unification, we can generalize the rule for typing the application:

$$\frac{s :: \tau_1, t :: \tau_2}{(s\ t) :: \sigma(\alpha)} \quad \text{if } \sigma \text{ is a mgu of } \tau_1 \doteq \tau_2 \to \alpha \text{ and } \alpha \text{ is a fresh type variable}$$

However, this rule is not sufficient for typing expressions with binders: consider the typing of an abstraction $\lambda x.s$. To type $\lambda x.s$, first the body s must be typed, and then a corresponding function type must be constructed. For example, in $\lambda x.\texttt{True}$ the body `True` can be typed with `Bool` and then the abstraction is typed with type $\alpha \to \texttt{True}$. But this case is too simple, because x does not occur in the body. A better rule is:

$$\frac{\text{Typing of } s \text{ with assumption } x \text{ is of type } \tau \text{ results in } s :: \tau'}{\lambda x.s :: \tau \to \tau'}$$

But now the correct type τ of x has to be guessed what does not lead to a deterministic algorithm. For this reason, one starts with a most general type for x and then computes the correct type by solving type equations using unification. To make this more efficient, we can do all typing steps, collect the necessary type equations and try to solve them all at once at the end using the unification algorithm.

The data structure of the typing has to be adapted to keep the equation. Besides the type, we also have equations E. The schema of the abstraction rule also shows that we need type assumptions, so they will also be part of our data structure. The used notation (also called a *typing judgement*) is:

$$\Gamma \vdash s :: \tau, E.$$

Its meaning is: Given the assumptions Γ, the type τ with unification equations E can be derived for expression s.

The assumption Γ can contain already known types: these are types for supercombinators and for data constructors. We use universally quantified types for these types to express that these types can be renamed. For example, for `map`, we could include the assumption: `map` $:: \forall a, b.(a \to b) \to [a] \to [b]$. Constructor applications are typed like nested applications (e.g. `Cons True Nil` is treated as ((`Cons True`) `Nil`). So data constructors are treated like constants. The type assumption for `Cons` is `Cons` $:: \forall a.a \to [a] \to [a]$.

We now list all the typing rules for KFPTSP+seq-expressions, where types of supercombinators must be included in the assumptions.

Axiom for variables:

$$\text{(AxV)}\ \frac{}{\Gamma \cup \{x :: \tau\} \vdash x :: \tau, \emptyset}$$

Axiom for constructors:

$$\text{(AxC)}\ \frac{}{\Gamma \cup \{c :: \forall\alpha_1 \ldots \alpha_n.\tau\} \vdash c :: \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n], \emptyset}\ \text{where } \beta_i \text{ are fresh type variables}$$

Axiom for supercombinators:

$$\text{(AxSC)}\ \frac{}{\Gamma \cup \{SC :: \forall\alpha_1 \ldots \alpha_n.\tau\} \vdash SC :: \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n], \emptyset}\ \text{where } \beta_i \text{ is a fresh type variable}$$

Rule for applications:

$$\text{(RApp)}\ \frac{\Gamma \vdash s :: \tau_1, E_1 \quad \text{and} \quad \Gamma \vdash t :: \tau_2, E_2}{\Gamma \vdash (s\ t) :: \alpha, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha\}}\ \text{where } \alpha \text{ is a fresh type variable}$$

Rule for `seq`:

$$\text{(RSeq)}\ \frac{\Gamma \vdash s :: \tau_1, E_1 \quad \text{and} \quad \Gamma \vdash t :: \tau_2, E_2}{\Gamma \vdash (\texttt{seq}\ s\ t) :: \tau_2, E_1 \cup E_2}$$

Rule for abstractions:

$$\text{(RAbs)}\ \frac{\Gamma \cup \{x :: \alpha\} \vdash s :: \tau, E}{\Gamma \vdash \lambda x.s :: \alpha \to \tau, E}\ \text{where } \alpha \text{ is a fresh type variable}$$

Rule for `case`:

$$\text{(RCase)}\ \frac{\begin{array}{l}\Gamma \vdash s :: \tau, E \\ \text{for all } i = 1, \ldots, m: \\ \quad \Gamma \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,\mathrm{ar}(c_i)} :: \alpha_{i,\mathrm{ar}(c_i)}\} \vdash (c_i\ x_{i,1}\ \ldots\ x_{i,\mathrm{ar}(ci)}) :: \tau_i, E_i \\ \text{for all } i = 1, \ldots, m: \\ \quad \Gamma \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,\mathrm{ar}(c_i)} :: \alpha_{i,\mathrm{ar}(c_i)}\} \vdash t_i :: \tau_i', E_i'\end{array}}{\Gamma \vdash \left(\begin{array}{l}\texttt{case}_T\ s\ \texttt{of}\{ \\ \quad (c_1\ x_{1,1}\ \ldots\ x_{1,\mathrm{ar}(c_1)}) \to t_1; \\ \quad \ldots; \\ \quad (c_m\ x_{m,1}\ \ldots\ x_{m,\mathrm{ar}(c_m)}) \to t_m\}\end{array}\right) :: \alpha, E'}$$

$$\text{where } E' = E \cup \bigcup_{i=1}^{m} E_i \cup \bigcup_{i=1}^{m} E_i' \cup \bigcup_{i=1}^{m} \{\tau \doteq \tau_i\} \cup \bigcup_{i=1}^{m} \{\alpha \doteq \tau_i'\}$$

and $\alpha_{i,j}, \alpha$ are fresh type variables

?? 2 explains how these typing rules are used to type a given expression s.

Algorithm 2: Type Inference of KFPTS+seq-Expressions

Let s be a closed KFPTS+seq-expression, where the types of all supercombinators and all constructors occurring in s are known.

1. Start with assumption Γ that contains types for the constructors and the supercombinators.
2. Derive $\Gamma \vdash s :: \tau, E$ with the typing rules.
3. Solve E with unification.
4. If unification ends with Fail, then s is not typeable; otherwise let σ be an mgu of E. Then the type of s is $s :: \sigma(\tau)$.

To optimize the type inference algorithm, the following rule can be added which allows unification to be performed as an intermediate step.

Type computation:

$$(\text{RUnif})\ \frac{\Gamma \vdash s :: \tau, E}{\Gamma \vdash s :: \sigma(\tau), E_\sigma}$$

where E_σ is the solved equation system E and σ is the unifier extracted from E_σ

Definition 5.3.1. *A* KFPTSP+seq-*expression s is* well-typed *iff it can be typed by* **??** *2.*

We consider some examples.

Example 5.3.2. *Consider the constructor application* `Cons True Nil`*. The assumption contains the types of* `Cons`*,* `Nil` *and* `True`*:*

$$\Gamma_0 = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} :: \forall a.[a], \texttt{True} :: \texttt{Bool}\}.$$

Typing of `Cons True Nil` *constructs the derivation tree bottom-up. The first step is*

$$(RA_{PP})\ \frac{\Gamma_0 \vdash (\texttt{Cons True}) :: \tau_1, E_1, \quad \Gamma_0 \vdash \texttt{Nil} :: \tau_2, E_2}{\Gamma_0 \vdash \texttt{Cons True Nil} :: \alpha_4, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha_4\}}$$

The exact types τ_1, τ_2 and the equation systems E_1, E_2 will be computed in the next step. In total the rule for applications has to be applied twice and the axiom for constants has to be applied three times. The complete derivation tree is:

$$\dfrac{\dfrac{\dfrac{}{\Gamma_0 \vdash \texttt{Cons} :: \alpha_1 \to [\alpha_1] \to [\alpha_1], \emptyset}(AxC) \quad \dfrac{}{\Gamma_0 \vdash \texttt{True} :: \texttt{Bool}, \emptyset}(AxC)}{\Gamma_0 \vdash (\texttt{Cons True}) :: \alpha_2, \{\alpha_1 \to [\alpha_1] \to [\alpha_1] \doteq \texttt{Bool} \to \alpha_2\}}(RA_{PP}) \quad \dfrac{}{\Gamma_0 \vdash \texttt{Nil} :: [\alpha_3], \emptyset}(AxC)}{\Gamma_0 \vdash (\texttt{Cons True Nil}) :: \alpha_4, \{\alpha_1 \to [\alpha_1] \to [\alpha_1] \doteq \texttt{Bool} \to \alpha_2, \alpha_2 \doteq [\alpha_3] \to \alpha_4\}}(RA_{PP})$$

To compute the type of (`Cons True Nil`)*, the equations have to unified. This results in:*

$$\sigma = \{\alpha_1 \mapsto \texttt{Bool}, \alpha_2 \mapsto [\texttt{Bool}] \to [\texttt{Bool}], \alpha_3 \mapsto \texttt{Bool}, \alpha_4 \mapsto [\texttt{Bool}]\}$$

and thus $(\texttt{Cons True Nil}) :: \sigma(\alpha_4) = [\texttt{Bool}]$.

Example 5.3.3. *The expression* $\Omega := (\lambda x.(x\ x))\ (\lambda y.(y\ y))$ *is* not *well-typed. The assumption is empty, since no constructors or supercombinators occur. The derivation tree is:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}(AxV), \dfrac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}(AxV)}{\{x :: \alpha_2\} \vdash (x\ x) :: \alpha_4, \{\alpha_2 \doteq \alpha_2 \to \alpha_4\}}(RAPP)}{\emptyset \vdash (\lambda x.(x\ x)) :: \alpha_2 \to \alpha_4, \{\alpha_2 \doteq \alpha_2 \to \alpha_4\}}(RABS), \dfrac{\dfrac{\dfrac{}{\{y :: \alpha_3\} \vdash y :: \alpha_3, \emptyset}(AxV), \dfrac{}{\{y :: \alpha_3\} \vdash y :: \alpha_3, \emptyset}(AxV)}{\{y :: \alpha_3\} \vdash (y\ y) :: \alpha_5, \{\alpha_3 \doteq \alpha_3 \to \alpha_5\}}(RAPP)}{\emptyset \vdash (\lambda y.(y\ y)) :: \alpha_3 \to \alpha_5, \{\alpha_3 \doteq \alpha_3 \to \alpha_5\}}(RABS)}{\emptyset \vdash (\lambda x.(x\ x))\ (\lambda y.(y\ y)) :: \alpha_1, \{\alpha_2 \doteq \alpha_2 \to \alpha_4, \alpha_3 \doteq \alpha_3 \to \alpha_5, \alpha_2 \to \alpha_4 \doteq (\alpha_3 \to \alpha_5) \to \alpha_1\}}(RAPP)$$

Unification fails, since:

$$(\textit{OccursCheck})\ \frac{\{\alpha_2 \doteq \alpha_2 \to \alpha_4, \ldots\}}{\textit{Fail}}$$

Exercise 5.3.4. *Show that the fixpoint supercombinator*

$$Y := \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

is not well-typed.

Example 5.3.5. *Suppose that supercombinators* `map` *and* `length` *are already typed. We type the expression*

$$t := \lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys) \to \texttt{map length}\ ys\}$$

We start with the assumption

$$\begin{aligned}\Gamma_0 = \{&\texttt{map} :: \forall a, b.(a \to b) \to [a] \to [b],\\ &\texttt{length} :: \forall a.[a] \to \texttt{Int},\\ &\texttt{Nil} :: \forall a.[a]\\ &\texttt{Cons} :: \forall a.a \to [a] \to [a]\}\end{aligned}$$

To construct the derivation tree, we start at the bottom of the tree and build it from the bottom up. So the first rule (at the bottom of the derivation tree) is (RABS)*, since* t *is an abstraction:*

$$(RABS)\ \frac{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to 1; (\texttt{Cons}\ y\ ys) \to \texttt{map length}\ ys\} :: \tau, E}{\Gamma_0 \vdash \alpha_1 \to \tau, E}$$

At this point type τ *and type equations* E *are not known. They will be filled in during the*

construction of the tree. The next step is to show the premises, i.e. rule (RCase) *has to be applied. For space reasons we separate the derivation tree and its labeling:*

$$\dfrac{\dfrac{\dfrac{}{B_3}\,(AxV), \dfrac{}{B_4}\,(AxC), \dfrac{\dfrac{\dfrac{}{B_8}\,(AxC), \dfrac{}{B_9}\,(AxV)}{B_6}\,(RApp), \dfrac{}{B_7}\,(AxV)}{B_5}\,(RApp)}{B_2}\,(RCase), \dfrac{}{B_{10}}\,(AxC), \dfrac{\dfrac{\dfrac{}{B_{14}}\,(AxSC), \dfrac{}{B_{15}}\,(AxSC)}{B_{12}}\,(RApp), \dfrac{}{B_{13}}\,(AxV)}{B_{11}}\,(RApp)}{B_1}\,(RAbs)$$

The labels are:

$$\begin{aligned}
B_1 =& \Gamma_0 \vdash t :: \alpha_1 \to \alpha_{13},\\
&\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7,\\
&(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12},\\
&\alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \}\\
B_2 =& \Gamma_0 \cup \{xs :: \alpha_1\} \vdash\\
&\quad \texttt{case}_{List}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys) \to \texttt{map}\ \texttt{length}\ ys\} :: \alpha_{13},\\
&\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7,\\
&(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12},\\
&\alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \}\\
B_3 =& \Gamma_0 \cup \{xs :: \alpha_1\} \vdash\ xs :: \alpha_1, \emptyset\\
B_4 =& \Gamma_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: [\alpha_2], \emptyset\\
B_5 =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{Cons}\ y\ ys) :: \alpha_7,\\
&\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7\}\\
B_6 =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{Cons}\ y) :: \alpha_6,\\
&\quad \{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6\}\\
B_7 =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash ys :: \alpha_4, \emptyset\\
B_8 =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset\\
B_9 =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash y :: \alpha_3, \emptyset\\
B_{10} =& \Gamma_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: [\alpha_{14}], \emptyset\\
B_{11} =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{map}\ \texttt{length})\ ys :: \alpha_{12},\\
&\quad \{(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12}\}\\
B_{12} =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{map}\ \texttt{length}) :: \alpha_{11},\\
&\quad \{(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}\}\\
B_{13} =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash ys :: \alpha_4, \emptyset
\end{aligned}$$

$$B_{14} = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \texttt{map} :: (\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9], \emptyset$$
$$B_{15} = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \texttt{length} :: [\alpha_{10}] \to \texttt{Int}, \emptyset$$

To compute the type of t, the type equations

$$\begin{array}{l}\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7,\\ (\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12},\\ \alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}\}\end{array}$$

have to be unified. This results in the mgu:

$$\begin{array}{rcl}\sigma & = & \{\alpha_1 \mapsto [[\alpha_{10}]], \alpha_2 \mapsto [\alpha_{10}], \alpha_3 \mapsto [\alpha_{10}], \alpha_4 \mapsto [[\alpha_{10}]], \alpha_5 \mapsto [\alpha_{10}],\\ & & \alpha_6 \mapsto [[\alpha_{10}]] \to [[\alpha_{10}]], \alpha_7 \mapsto [[\alpha_{10}]], \alpha_8 \mapsto [\alpha_{10}], \alpha_9 \mapsto \texttt{Int},\\ & & \alpha_{11} \mapsto [[\alpha_{10}]] \to [\texttt{Int}], \alpha_{12} \mapsto [\texttt{Int}], \alpha_{13} \mapsto [\texttt{Int}], \alpha_{14} \mapsto \texttt{Int}\}\end{array}$$

Finally, we derive $t :: \sigma(\alpha_1 \to \alpha_{13}) = [[\alpha_{10}]] \to [\texttt{Int}]$.

Example 5.3.6. *The supercombinator* `const` *is defined as:*

```
const :: a -> b -> a
const x y = x
```

The expression $\lambda x.$`const` $(x$ `True`$)$ $(x$ `'A'`$)$ *is not well-typed: Let*

$$\Gamma_0 = \{\texttt{const} :: \forall a, b.a \to b \to a, \texttt{True} :: \texttt{Bool}, \texttt{'A'} :: \texttt{Char}\}$$

and $\Gamma_1 = \Gamma_0 \cup \{x :: \alpha_1\}$. *The derivation tree is:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash \texttt{const} :: \alpha_2 \to \alpha_3 \to \alpha_2, \emptyset}(AxSC), \dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash x :: \alpha_1}(AxV), \dfrac{}{\Gamma_1 \vdash \texttt{True} :: \texttt{Bool}}(AxC)}{\Gamma_1 \vdash (x\ \texttt{True}) :: \alpha_4, E_1}(RApp)}{\Gamma_1 \vdash \texttt{const}\ (x\ \texttt{True}) :: \alpha_5, E_2}(RApp), \dfrac{\dfrac{}{\Gamma_1 \vdash x :: \alpha_1}(AxV), \dfrac{}{\Gamma_1 \vdash \texttt{'A'} :: \texttt{Char}}(AxC)}{\Gamma_1 \vdash (x\ \texttt{'A'}) :: \alpha_6, E_3}(RApp)}{\Gamma_1 \vdash \texttt{const}\ (x\ \texttt{True})\ (x\ \texttt{'A'}) :: \alpha_7, E_4}(RApp)}{\Gamma_0 \vdash \lambda x.\texttt{const}\ (x\ \texttt{True})\ (x\ \texttt{'A'}) :: \alpha_1 \to \alpha_7, E_4}(RAbs)$$

with

$$\begin{array}{rcl}E_1 & = & \{\alpha_1 \doteq \texttt{Bool} \to \alpha_4\}\\ E_2 & = & \{\alpha_1 \doteq \texttt{Bool} \to \alpha_4, \alpha_2 \to \alpha_3 \to \alpha_2 \doteq \alpha_4 \to \alpha_5\}\\ E_3 & = & \{\alpha_1 \doteq \texttt{Char} \to \alpha_6\}\\ E_4 & = & \{\alpha_1 \doteq \texttt{Bool} \to \alpha_4, \alpha_2 \to \alpha_3 \to \alpha_2 \doteq \alpha_4 \to \alpha_5, \alpha_1 \doteq \texttt{Char} \to \alpha_6, \alpha_5 \doteq \alpha_6 \to \alpha_7\}\end{array}$$

Unification fails, since the two equations for α_1 *are not unifiable (*`Bool` *cannot match* `Char`*).*

The last example shows, that the type inference algorithm treats lambda-bound variables as monomorphically typed: For an abstraction $\lambda x.s$, all occurrences of x in s must be of the same (monomorphic) type. Note that the expression $\lambda x.$`const` $(x$ `True`$)$ $(x$ `'A'`$)$ can be applied to useful arguments, like a function that ignores its argument and returns a constant value

Since lambda-bound variables are monomorphically typed (also pattern variables in `case`-alternatives), in programming languages like Haskell one speaks of `let`-*polymorphism* , since only `let`-bound variables are allowed to be polymorphically typed. In KFPTSP+seq there is no `let`, but the supercombinators play a similar role.

Attempting to adapt the type system so that the expression from above is typeable, would require expressing that for x only such abstractions are allowed that return the same result type for arbitrary argument types. Our type system cannot express such conditions, since they do not conform to the set semantics $\mathtt{sem}(\cdot)$. Our type system uses so-called *predicative polymorphism*, since type variables represent monomorphic types, in type systems with *impredicative polymorphism* type variables can represent polymorphic types.

In extended type systems the above expression could be typed as:

```
(\x -> const (x True) (x 'A'))::(forall b.(forall a. a -> b) -> b)
```

The inner universal quantifier is not allowed in our type syntax.

5.4. Typing of Non-Recursive Supercombinators

We consider the typing of non-recursive supercombinators, i.e. supercombinators that do not call themselves (also not through a chain of calls). Let $\mathcal{SC}$ be a set of supercombinators. For $SC_i, SC_j \in \mathcal{SC}$, let $SC_i \preceq SC_j$ iff the right-hand side of the definition of SC_j contains the supercombinator SC_i. Let $\preceq^+$ be the transitive closure of $\preceq$ (and $\preceq^*$ be the reflexive-transitive closure).

A supercombinator SC_i is *directly recursive* iff $SC_i \preceq SC_i$ holds, it is *recursive* iff $SC_i \preceq^+ SC_i$ holds. A subset $\{SC_1, \ldots, SC_m\} \subseteq \mathcal{SC}$ of supercombinators is mutually recursive if $SC_i \preceq^+ SC_j$ for all $i, j \in \{1, \ldots, m\}$.

Non-recursive supercombinators can be typed analogously to abstractions (other supercombinators in the right-hand side of the defining equation have to be typed already). We write $\Gamma \vdash_T SC :: \tau$, if supercombinator SC can be typed τ under the assumption Γ. The rule for non-recursive supercombinators is

$$\text{(RSC1)}\ \frac{\Gamma \cup \{x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{\Gamma \vdash_T SC :: \forall \mathcal{X}.\sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau)}$$

if σ is the solution of E, $SC\ x_1\ \ldots\ x_n = s$ is the definition of SC, SC is non-recursive, and $\mathcal{X} = Vars(\sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau))$

Example 5.4.1. *We type the composition, defined as*

```
(.) f g x = f (g x)
```

The assumption Γ_0 is empty, since in the right-hand side of the definition of `(.)` *no constructors or other supercombinators occur. Let $\Gamma_1 = \{f :: \alpha_1, g :: \alpha_2, x :: \alpha_3\}$. A type derivation for* `(.)` *is:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash f :: \alpha_1, \emptyset,}\,(AxV) \quad \dfrac{\dfrac{}{\Gamma_1 \vdash g :: \alpha_2, \emptyset,}\,(AxV) \quad \dfrac{}{\Gamma_1 \vdash x :: \alpha_3, \emptyset}\,(AxV)}{\Gamma_1 \vdash (g\ x) :: \alpha_5, \{\alpha_2 \doteq \alpha_3 \to \alpha_5\}}\,(RApp)}{\Gamma_1 \vdash (f\ (g\ x)) :: \alpha_4, \{\alpha_2 \doteq \alpha_3 \to \alpha_5, \alpha_1 = \alpha_5 \to \alpha_4\}}\,(RApp)}{\emptyset \vdash_T (.) :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4) = (\alpha_5 \to \alpha_4) \to (\alpha_3 \to \alpha_5) \to \alpha_3 \to \alpha_4}\,(RSC1)$$

Here $\sigma = \{\alpha_2 \mapsto \alpha_3 \to \alpha_5, \alpha_1 \mapsto \alpha_5 \to \alpha_4\}$. *The derived type can now be universally quantified and be used for typing other supercombinator that use the it.*

$$(.) :: \forall a, b, c.(a \to b) \to (c \to a) \to c \to b$$

5.5. Typing of Recursive Supercombinators

To type recursive supercombinators (or a group of mutually recursive supercombinators) the obstacle is, that for typing the right-hand side of the supercombinator definition, the type of the to-be-typed supercombinator would be required (but it is not available at this point of time). The solution is to add (a most general) assumption about the type of the to-be-typed supercombinators and then use type derivation and unification to refine the type. This is iterated until a so-called *consistent type assumption* is found.

5.5.1. The Iterative Type Inference Algorithm

The iterative type inference algorithm starts with most general assumption, computes new assumptions using the type derivation rules, and then compares the derived assumptions with the used ones. If they are changed, then the new assumptions are used and a new iteration starts. If the derived assumptions coincides with the used ones, then the assumptions are consistent. The algorithm has found a type and stops.

The typing rule for supercombinators is similar to the non-recursive case:

$$(\text{SCR}_{\text{EC}})\ \frac{\Gamma \cup \{x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{\Gamma \vdash_T SC :: \sigma(\alpha_1 \to \ldots \alpha_n \to \tau)}\ \begin{array}{l}\text{if } SC\ x_1\ \ldots\ x_n = s \text{ is the definition}\\ \text{of } SC,\ \sigma \text{ solution of } E\end{array}$$

The difference is that the assumption Γ already contains a type assumption for SC (or all supercombinators of a mutually recursive set of supercombinators, resp.) (see **??** 3).

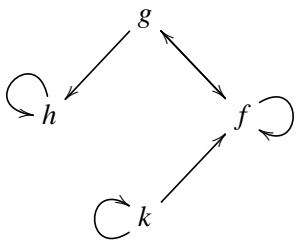
Given a whole program with a set of supercombinator definitions, one starts with a dependency analysis that computes the strongly connected components in the call graph. If $\simeq$ is the equivalence relation corresponding to $\preceq^*$, then the strongly connected components of the call graph are the equivalence classes of $\simeq$. In a next step the smallest equivalence classes w.r.t. $\simeq$ are typed (they do not depend on other supercombinators) and then the supercombinators are typed along the

dependency graph. Of course, all supercombinators of the same equivalence class are mutually recursive and they must be typed together.

Let the supercombinators f, g, h, k be defined as:

$$\begin{array}{lcl} f\ x\ y & = & \texttt{if}\ x \leq 1\ \texttt{then}\ y\ \texttt{else}\ f\ (x-y)\ (y + (g\ x)) \\ g\ x & = & \texttt{if}\ x = 0\ \texttt{then}\ (f\ 1\ x) + (h\ 2)\ \texttt{else}\ 10 \\ h\ x & = & \texttt{if}\ x = 1\ \texttt{then}\ 0\ \texttt{else}\ h\ (x-1) \\ k\ x\ y & = & \texttt{if}\ x = 1\ \texttt{then}\ y\ \texttt{else}\ k\ (x-1)\ (y + (f\ x\ y)) \end{array}$$

The call graph is:



The equivalence classes (ordered) are $\{h\} \preceq^+ \{f, g\} \preceq^+ \{k\}$. Thus, first h has to be typed, then f and g are typed together, and finally k is typed.

The *iterative type inference algorithm* is given in **??** 3.

The assumption at the beginning in step (2) is the most general assumption that can be made, since the polymorphic type $\forall a.a$ represents any arbitrary type ($\texttt{sem}(a)$ is the set of all monomorphic types).

The following properties hold for the iterative type inference algorithm:

- The computed types are unique up to renaming for each iteration and thus: if the algorithm terminates, then the types of the supercombinators are unique.
- In each step the newly computed types are more specific or remains the same (in terms of the semantics, the set of monomorphic types is contained in the previous one).
- If the algorithm does not terminate, then no polymorphic type for the supercombinators exists. This holds, since the types become more specific in each step and the algorithm starts with the full set.
- The iterative type inference algorithm computes the greatest fixpoint w.r.t. `sem`: the algorithm starts with the full set and restricts the set until there is no change. Mathematically, suppose that F is the operator that performs one iteration of the algorithm on the set of monomorphic types. If the algorithm stops with set S, then $F(S) = S$ (so S is a fixpoint) and S is the largest set M such that $F(M) = M$. This shows, that the iterative type inference algorithm computes the most general polymorphic type.

Algorithm 3: Iterative Type Inference Algorithm

The input is a set of mutually recursive supercombinators $SC_1, \ldots, SC_m$, where all other used supercombinators are already typed.

1. The start assumption Γ consists of the types of the constructor and the already typed supercombinators.
2. $\Gamma_0 := \Gamma \cup \{SC_1 :: \forall\alpha_1.\alpha_1, \ldots, SC_m :: \forall\alpha_m.\alpha_m\}$ and $j = 0$.
3. For each supercombinator SC_i (with $i = 1, \ldots, m$) apply the rule (SCRec) and assumption Γ_j top type SC_i.
4. If all m type derivations were successful, then

   $$\Gamma_j \vdash_T SC_1 :: \tau_1, \ldots, \Gamma_j \vdash_T SC_m :: \tau_m.$$

   Now universally quantify all the types τ_i by quantifying all type variables. W.l.o.g. assume this gives $SC_1 :: \forall\mathcal{X}_1.\tau_1, \ldots, SC_m :: \forall\mathcal{X}_m.\tau_m$. Let

   $$\Gamma_{j+1} := \Gamma \cup \{SC_1 :: \forall\mathcal{X}_1.\tau_1, \ldots, SC_m :: \forall\mathcal{X}_m.\tau_m\}$$

5. If $\Gamma_j \neq \Gamma_{j+1}$ (where equality up to renaming of type variables is meant), then set $j := j + 1$ and proceed with step (3). Otherwise ($\Gamma_j = \Gamma_{j+1}$) the assumptions Γ_j are consistent. The types of SC_i are in Γ_j.

If a *Fail* occurs in any unification, then $SC_1, \ldots, SC_m$ are not typeable.

5.5.2. Examples and Properties

Example 5.5.1. *We type the supercombinator* `length` *with definition:*

$$\texttt{length}\ xs = \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to 0; (y : ys) \to 1 + \texttt{length}\ ys\}$$

The type assumption for the constructors and $(+)$ *is:*

$$\Gamma = \{\texttt{Nil} :: \forall a.[a], (:) :: \forall a.a \to [a] \to [a], 0 :: \texttt{Int}, 1 :: \texttt{Int}, (+) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}\}.$$

The iterative type inference algorithm (for $j = 0$*) assumes the most general type for* `length`*, i.e.* $\Gamma_0 = \Gamma \cup \{\texttt{length} :: \forall\alpha.\alpha\}$ *and applies rule* (*SCRec*) *to type* `length` *in the first iteration:*

$$
\dfrac{\dfrac{\begin{array}{ll}(a) & \Gamma_0 \cup \{xs :: \alpha_1\} \vdash xs :: \tau_1, E_1\\ (b) & \Gamma_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: \tau_2, E_2\\ (c) & \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\} \vdash (y : ys) :: \tau_3, E_3\\ (d) & \Gamma_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \tau_4, E_4\\ (e) & \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}\} \vdash (1 + \texttt{length}\ ys) :: \tau_5, E_5\end{array}}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash (\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to 0; (y : ys) \to 1 + \texttt{length}\ xs\}) :: \alpha_3,\ E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \doteq \tau_2, \tau_1 \doteq \tau_3, \alpha_3 \doteq \tau_4, \alpha_3 \doteq \tau_5\}}\ (RC_{ASE})}{\Gamma_0 \vdash_T \texttt{length} :: \sigma(\alpha_1 \to \alpha_3)}\ (SCR_{EC})
$$

where σ is the solution of $E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \doteq \tau_2, \tau_1 \doteq \tau_3, \alpha_3 \doteq \tau_4, \alpha_3 \doteq \tau_5\}$

We show the premises (a) to (e) separately (types $\tau_1, \ldots, \tau_5$ and equation $E_1, \ldots, E_5$ are computed in these derivations):

(a): $\dfrac{}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash xs :: \alpha_1, \emptyset}\ (AxV)$

I.e, $\tau_1 = \alpha_1$ and $E_1 = \emptyset$

(b): $\dfrac{}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: [\alpha_6], \emptyset}\ (AxC)$

I.e., $\tau_2 = [\alpha_6]$ and $E_2 = \emptyset$

(c)
$$
\dfrac{\dfrac{\dfrac{}{\Gamma_0' \vdash (:) :: \alpha_9 \to [\alpha_9] \to [\alpha_9], \emptyset}\ (AxC) \quad \dfrac{}{\Gamma_0' \vdash y :: \alpha_4, \emptyset}\ (AxV)}{\Gamma_0' \vdash ((:)\ y) :: \alpha_8, \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8\}}\ (RA_{PP}) \quad \dfrac{}{\Gamma_0' \vdash ys :: \alpha_5, \emptyset}\ (AxV)}{\Gamma_0' \vdash (y : ys) :: \alpha_7, \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\}}\ (RA_{PP})
$$

where $\Gamma_0 = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$

I.e., $\tau_3 = \alpha_7$ and $E_3 = \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\}$

(d) $\dfrac{}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \texttt{Int}, \emptyset}\ (AxC)$

I.e., $\tau_4 = \texttt{Int}$ and $E_4 = \emptyset$

(e)
$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_0' \vdash (+) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}, \emptyset}\ (AxC) \quad \dfrac{}{\Gamma_0' \vdash 1 :: \texttt{Int}, \emptyset}\ (AxC)}{\Gamma_0' \vdash ((+)\ 1) :: \alpha_{11}, \{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}\}}\ (RA_{PP}) \quad \dfrac{\dfrac{}{\Gamma_0' \vdash \texttt{length} :: \alpha_{13}, \emptyset}\ (AxSC) \quad \dfrac{}{\Gamma_0' \vdash (ys) :: \alpha_5, \emptyset}\ (AxV)}{\Gamma_0' \vdash (\texttt{length}\ ys) :: \alpha_{12}, \{\alpha_{13} \doteq \alpha_5 \to \alpha_{12}\}}\ (RA_{PP})}{\Gamma_0' \vdash (1 + \texttt{length}\ ys) :: \alpha_{10}, \{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\}}\ (RA_{PP})
$$

where $\Gamma_0' = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$

I.e., $\tau_5 = \alpha_{10}$ and $E_5 = \{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\}$

Substituting the values, the application of rule (SCR_{EC}) is:

$$\text{(SCR}_{\text{EC}}\text{)}\ \frac{\begin{array}{l}\Gamma_0 \cup \{xs :: \alpha_1\} \vdash (\mathtt{case}_{\mathtt{List}}\ xs\ \mathtt{of}\ \{\mathtt{Nil} \to 0; (y : ys) \to 1 + \mathtt{length}\ xs\}) :: \alpha_3,\\ \quad \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\}\\ \cup\{\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \doteq \mathtt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\}\\ \cup\{\alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \mathtt{Int}, \alpha_3 \doteq \alpha_{10}\}\end{array}}{\begin{array}{c}\Gamma_0 \vdash_T \mathtt{length} :: \sigma(\alpha_1 \to \alpha_3)\\ \textit{where } \sigma \textit{ is the solution of}\\ \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7,\\ \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \doteq \mathtt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10},\\ \alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \mathtt{Int}, \alpha_3 \doteq \alpha_{10}\}\end{array}}$$

Unification is successful and computes the unifier

$$\{\alpha_1 \mapsto [\alpha_9], \alpha_3 \mapsto \mathtt{Int}, \alpha_4 \mapsto \alpha_9, \alpha_5 \mapsto [\alpha_9], \alpha_6 \mapsto \alpha_9, \alpha_7 \mapsto [\alpha_9], \alpha_8 \mapsto [\alpha_9] \to [\alpha_9],$$
$$\alpha_{10} \mapsto \mathtt{Int}, \alpha_{11} \mapsto \mathtt{Int} \to \mathtt{Int}, \alpha_{12} \mapsto \mathtt{Int}, \alpha_{13} \mapsto [\alpha_9] \to \mathtt{Int}\}$$

The newly computed type of `length` *is* $\sigma(\alpha_1 \to \alpha_3\} = [\alpha_9] \to \mathtt{Int}$ *and*

$$\Gamma_1 = \Gamma \cup \{\mathtt{length} :: \forall\alpha.[\alpha] \to \mathtt{Int}\}.$$

Since $\Gamma_0 \neq \Gamma_1$*, the next iteration using* Γ_1 *has to started.*

To shorten the presentation, we list only the substep that is different from the first iteration: The difference is in part (e)*: instead of*

$$\text{(AxSC)}\ \frac{}{\Gamma_0' \vdash \mathtt{length} :: \alpha_{13}, \emptyset}$$

now

$$\text{(AxSC)}\ \frac{}{\Gamma_1' \vdash \mathtt{length} :: [\alpha_{13}] \to \mathtt{Int}, \emptyset}$$

is derived. Adjusting all equation systems and replacing Γ_0 *with* Γ_1 *results in the following use of rule* (SCR_{EC})*:*

$$\frac{\begin{array}{l}\Gamma_1 \cup \{xs :: \alpha_1\} \vdash (\mathtt{case}_{\mathtt{List}}\ xs\ \mathtt{of}\ \{\mathtt{Nil} \to 0; (y : ys) \to 1 + \mathtt{length}\ xs\}) :: \alpha_3,\\ \quad \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\}\\ \cup\{\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \doteq \mathtt{Int} \to \alpha_{11}, : [\alpha_{13}] \to \mathtt{Int} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\}\\ \cup\{\alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \mathtt{Int}, \alpha_3 \doteq \alpha_{10}\}\end{array}}{\begin{array}{c}\Gamma_1 \vdash_T \mathtt{length} :: \sigma(\alpha_1 \to \alpha_3)\\ \textit{where } \sigma \textit{ is the solution of}\\ \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7,\\ \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \doteq \mathtt{Int} \to \alpha_{11}, [\alpha_{13}] \to \mathtt{Int} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10},\\ \alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \mathtt{Int}, \alpha_3 \doteq \alpha_{10}\}\end{array}}\ (SCR_{EC})$$

Unification computes the unifier σ:

$$\{\alpha_1 \mapsto [\alpha_9], \alpha_3 \mapsto \mathtt{Int}, \alpha_4 \mapsto \alpha_9, \alpha_5 \mapsto [\alpha_9], \alpha_6 \mapsto \alpha_9, \alpha_7 \mapsto [\alpha_9], \alpha_8 \mapsto [\alpha_9] \to [\alpha_9],$$
$$\alpha_{10} \mapsto \mathtt{Int}, \alpha_{11} \mapsto \mathtt{Int} \to \mathtt{Int}, \alpha_{12} \mapsto \mathtt{Int}, \alpha_{12} \mapsto \mathtt{Int}, \alpha_{13} \mapsto \alpha_9\}$$

Thus $\mathtt{length} :: \sigma(\alpha_1 \to \alpha_3) = [\alpha_9] \to \mathtt{Int}$ *and* $\Gamma_2 = \Gamma \cup \{\mathtt{length} :: \forall\alpha.[\alpha] \to \mathtt{Int}\}$. *Since* $\Gamma_1 = \Gamma_2$, *assumption* Γ_1 *is consistent and* $[\alpha] \to \mathtt{Int}$ *is the iterative type of* `length`.

5.5.2.1. Iterative Typing is More General than Haskell

Example 5.5.2. *We type the supercombinator* `g` *with definition*

```
g x = 1 : (g (g 'c'))
```

The assumptions on the constructors are $\Gamma = \{1 :: \mathtt{Int}, \mathtt{Cons} :: \forall a.a \to [a] \to [a], \mathtt{'c'} :: \mathtt{Char}\}$. *The iterative type inference algorithm starts with* $\Gamma_0 = \Gamma \cup \{\mathtt{g} :: \forall\alpha.\alpha\}$ *and types* `g` *with rule* (SCR_{EC}) *(let* $\Gamma_0' = \Gamma_0 \cup \{x :: \alpha_1\}$*):*

$$\frac{\dfrac{\dfrac{(AxC)\ \overline{\Gamma_0' \vdash \mathtt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}, \quad (AxC)\ \overline{\Gamma_0' \vdash 1 :: \mathtt{Int}, \emptyset}}{(RA_{PP})\ \Gamma_0' \vdash (\mathtt{Cons}\ 1) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \mathtt{Int} \to \alpha_3}, \quad \dfrac{(AxSC)\ \overline{\Gamma_0' \vdash \mathtt{g} :: \alpha_6, \emptyset}, \quad \dfrac{(AxSC)\ \overline{\Gamma_0' \vdash \mathtt{g} :: \alpha_8, \emptyset}, \quad (AxC)\ \overline{\Gamma_0' \vdash \mathtt{'c'} :: \mathtt{Char}, \emptyset}}{(RA_{PP})\ \Gamma_0' \vdash (\mathtt{g}\ \mathtt{'c'}) :: \alpha_7, \{\alpha_8 \doteq \mathtt{Char} \to \alpha_7\}}}{(RA_{PP})\ \Gamma_0' \vdash (\mathtt{g}\ (\mathtt{g}\ \mathtt{'c'})) :: \alpha_4, \{\alpha_8 \doteq \mathtt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4\}}}{(RA_{PP})\ \Gamma_0' \vdash \mathtt{Cons}\ 1\ (\mathtt{g}\ (\mathtt{g}\ \mathtt{'c'})) :: \alpha_2, \{\alpha_8 \doteq \mathtt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \mathtt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}}{\Gamma_0 \vdash_T \mathtt{g} :: \sigma(\alpha_1 \to \alpha_2) = \alpha_1 \to [\mathtt{Int}]}\ (SCR_{EC})$$

where $\sigma = \{\alpha_2 \mapsto [\mathtt{Int}], \alpha_3 \mapsto [\mathtt{Int}] \to [\mathtt{Int}], \alpha_4 \mapsto [\mathtt{Int}], \alpha_5 \mapsto \mathtt{Int}, \alpha_6 \mapsto \alpha_7 \to [\mathtt{Int}], \alpha_8 \mapsto \mathtt{Char} \to \alpha_7\}$ *is the unifier of*
$\{\alpha_8 \doteq \mathtt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \mathtt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$

Hence, we have $\Gamma_1 = \Gamma \cup \{\mathtt{g} :: \forall\alpha.\alpha \to [\mathtt{Int}]\}$. *The next iteration shows that* Γ_1 *is consistent (we omit the iteration). However, in Haskell, the interpreter cannot infer a type for the supercombinator* `g`*:*

```
Prelude> let g x = 1:(g(g 'c'))

<interactive>:1:13:
Couldn't match expected type ‘[t]’ against inferred type ‘Char’
      Expected type: Char -> [t]
      Inferred type: Char -> Char
    In the second argument of ‘(:)’, namely ‘(g (g 'c'))’
    In the expression: 1 : (g (g 'c'))
```

But, the Haskell-interpreter can verify the type if it is provided by the programmer:

```
let g::a -> [Int]; g x = 1:(g(g 'c'))
Prelude> :t g
g :: a -> [Int]
```

The reason is that in the latter case, the Haskell-interpreter only performs type checking, but no type inference, and uses the provided type of g *as universally quantified.*

5.5.2.2. Multiple Iterations are Required

Example 5.5.3. *We type the supercombinator* g *with definition*

```
g x = x : (g (g 'c'))
```

The assumption for the constructors is $\Gamma = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{'c'} :: \texttt{Char}\}$. *Iterative typing starts with* $\Gamma_0 = \Gamma \cup \{\texttt{g} :: \forall\alpha.\alpha\}$ *and types* g *with rule (SCRec) (let* $\Gamma_0' = \Gamma_0 \cup \{x :: \alpha_1\}$*):*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_0' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,{\scriptstyle (AxC)} \quad \dfrac{}{\Gamma_0' \vdash x :: \alpha_1, \emptyset}\,{\scriptstyle (AxV)}}{\Gamma_0' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}\,{\scriptstyle (RApp)} \quad \dfrac{\dfrac{}{\Gamma_0' \vdash \texttt{g} :: \alpha_6, \emptyset}\,{\scriptstyle (AxSC)}}{\Gamma_0' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_4, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4\}}\,{\scriptstyle (RApp)}}{\Gamma_0' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}\,{\scriptstyle (RApp)}}{\Gamma_0 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = \alpha_5 \to [\alpha_5]}\,{\scriptstyle (SCRec)}$$

where $\sigma = \{\alpha_1 \mapsto \alpha_5, \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [\alpha_5] \to [\alpha_5], \alpha_4 \mapsto [\alpha_5], \alpha_6 \mapsto \alpha_7 \to [\alpha_5], \alpha_8 \mapsto \texttt{Char} \to \alpha_7\}$ *is the solution of* $\{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$

Thus, we get $\Gamma_1 = \Gamma \cup \{\texttt{g} :: \forall\alpha.\alpha \to [\alpha]\}$. *Since* $\Gamma_0 \neq \Gamma_1$, *another iteration is necessary. Let* $\Gamma_1' = \Gamma_1 \cup \{x :: \alpha_1\}$*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,{\scriptstyle (AxC)} \quad \dfrac{}{\Gamma_1' \vdash x :: \alpha_1, \emptyset}\,{\scriptstyle (AxV)}}{\Gamma_1' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}\,{\scriptstyle (RApp)} \quad \dfrac{\dfrac{}{\Gamma_1' \vdash \texttt{g} :: \alpha_6 \to [\alpha_6], \emptyset}\,{\scriptstyle (AxSC)}}{\Gamma_1' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4\}}\,{\scriptstyle (RApp)}}{\Gamma_1' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}\,{\scriptstyle (RApp)}}{\Gamma_1 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = [\texttt{Char}] \to [[\texttt{Char}]]}\,{\scriptstyle (SCRec)}$$

where $\sigma = \{\alpha_1 \mapsto [\texttt{Char}], \alpha_2 \mapsto [[\texttt{Char}]], \alpha_3 \mapsto [[\texttt{Char}]] \to [[\texttt{Char}]], \alpha_4 \mapsto [[\texttt{Char}]], \alpha_5 \mapsto [\texttt{Char}], \alpha_6 \mapsto [\texttt{Char}], \alpha_7 \mapsto [\texttt{Char}], \alpha_8 \mapsto \texttt{Char}\}$
is the solution of $\{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$

Thus, we get $\Gamma_2 = \Gamma \cup \{\texttt{g} :: [\texttt{Char}] \to [[\texttt{Char}]]\}$. *Since* $\Gamma_1 \neq \Gamma_2$, *another iteration is necessary: Let* $\Gamma_2' = \Gamma_2 \cup \{x :: \alpha_1\}$*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_2' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,{\scriptstyle (AxC)} \quad \dfrac{}{\Gamma_2' \vdash x :: \alpha_1, \emptyset}\,{\scriptstyle (AxV)}}{\Gamma_2' \vdash (\texttt{Cons}\ x) :: \alpha_3, \{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3\}}\,{\scriptstyle (RApr)} \quad \dfrac{\dfrac{}{\Gamma_2' \vdash \texttt{g} :: [\texttt{Char}] \to [[\texttt{Char}]], \emptyset}\,{\scriptstyle (AxSC)} \quad \dfrac{\dfrac{}{\Gamma_2' \vdash \texttt{g} :: [\texttt{Char}] \to [[\texttt{Char}]], \emptyset}\,{\scriptstyle (AxSC)} \quad \dfrac{}{\Gamma_2' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}\,{\scriptstyle (AxC)}}{\Gamma_2' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7\}}\,{\scriptstyle (RApr)}}{\Gamma_2' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7, [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4\}}\,{\scriptstyle (RApr)}}{\begin{array}{l}\Gamma_2' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\ [\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7,\ \} \\ \qquad\qquad [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4, \\ \qquad\qquad \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \\ \qquad\qquad \alpha_3 \doteq \alpha_4 \to \alpha_2\end{array}}$$

$$\dfrac{\Gamma_2' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\ [\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7,\ \} \quad [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4,\ \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3,\ \alpha_3 \doteq \alpha_4 \to \alpha_2}{\begin{array}{c}\Gamma_2 \vdash_\Gamma \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) \\ \textit{where } \sigma \textit{ is the solution of} \\ \{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7, [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}\end{array}}\,{\scriptstyle (SCRec)}$$

Unification of the equation fails and thus `g` *is not typeable.*

Proposition 5.5.4. *The iterative type inference algorithm sometimes requires multiple iterations until a result (untyped / consistent assumption) is found.*

Proof. Example 5.5.3 provides an example showing that three iterations are necessary to derive that a supercombinator is untyped. An example that is typed and requires more iterations is given in Exercise 5.5.5). □

Exercise 5.5.5. *Apply the iterative type inference algorithm to the supercombinator* `fix` *with definition:*

```
fix f = f (fix f)
```

Show that the iterative type inference algorithm requires several iterations until a consistent assumption is derived.

5.5.2.3. Non-Termination of the Iterative Type Inference Algorithm

Example 5.5.6. *Let* `f` *and* `g` *be supercombinators with definitions:*

```
f = [g]
g = [f]
```

The supercombinators are mutually recursive (i.e. $\texttt{f} \simeq \texttt{g}$*) and thus the iterative type inference algorithm types them together.*

The assumption for the constructors is $\Gamma = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} : \forall a.a\}$. *The iterative type inference algorithm starts with* $\Gamma_0 = \Gamma \cup \{\texttt{f} :: \forall \alpha.\alpha, \texttt{g} :: \forall \alpha.\alpha\}$ *and applies the rule* (*SCRec*) *for* `f` *and* `g`*:*

$$\dfrac{\dfrac{\dfrac{(AxC)\ \dfrac{}{\Gamma_0 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}, \quad (AxSC)\ \dfrac{}{\Gamma_0 \vdash \texttt{g} :: \alpha_5}}{(RA_{PP})\ \dfrac{}{\Gamma_0 \vdash (\texttt{Cons g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3\}}}, \quad (AxC)\ \dfrac{}{\Gamma_0 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{(RA_{PP})\ \dfrac{}{\Gamma_0 \vdash [\texttt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}}}{(SCR_{EC})\ \dfrac{}{\Gamma_0 \vdash_T \texttt{f} :: \sigma(\alpha_1) = [\alpha_5]}}$$

$\sigma = \{\alpha_1 \mapsto [\alpha_5], \alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto [\alpha_5] \to [\alpha_5], \alpha_4 \mapsto \alpha_5\}$ *is a solution of* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

$$\dfrac{\dfrac{\dfrac{(AxC)\ \dfrac{}{\Gamma_0 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}, \quad (AxSC)\ \dfrac{}{\Gamma_0 \vdash \texttt{f} :: \alpha_5}}{(RA_{PP})\ \dfrac{}{\Gamma_0 \vdash (\texttt{Cons f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3\}}}, \quad (AxC)\ \dfrac{}{\Gamma_0 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{(RA_{PP})\ \dfrac{}{\Gamma_0 \vdash [\texttt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}}}{(SCR_{EC})\ \dfrac{}{\Gamma_0 \vdash_T \texttt{g} :: \sigma(\alpha_1) = [\alpha_5]}}$$

$\sigma = \{\alpha_1 \mapsto [\alpha_5], \alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto [\alpha_5] \to [\alpha_5], \alpha_4 \mapsto \alpha_5\}$ *is a solution of* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

Hence, we get $\Gamma_1 = \Gamma \cup \{\texttt{f} :: \forall a.[a], \texttt{g} :: \forall a.[a]\}$. *Since* $\Gamma_1 \neq \Gamma_0$ *a next iteration is required.*

$$\dfrac{\dfrac{\dfrac{(AxC)\ \dfrac{}{\Gamma_1 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}, \quad (AxSC)\ \dfrac{}{\Gamma_1 \vdash \texttt{g} :: [\alpha_5]}}{(RA_{PP})\ \dfrac{}{\Gamma_1 \vdash (\texttt{Cons g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3\}}}, \quad (AxC)\ \dfrac{}{\Gamma_1 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{(RA_{PP})\ \dfrac{}{\Gamma_1 \vdash [\texttt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}}}{(SCR_{EC})\ \dfrac{}{\Gamma_1 \vdash_T \texttt{f} :: \sigma(\alpha_1) = [[\alpha_5]]}}$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]], \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [[\alpha_5]] \to [[\alpha_5]], \alpha_4 \mapsto [\alpha_5]\}$ *is a solution of* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

$$\dfrac{\dfrac{\dfrac{(AxC)\ \dfrac{}{\Gamma_1 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}, \quad (AxSC)\ \dfrac{}{\Gamma_1 \vdash \texttt{f} :: [\alpha_5]}}{(RA_{PP})\ \dfrac{}{\Gamma_1 \vdash (\texttt{Cons f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3\}}}, \quad (AxC)\ \dfrac{}{\Gamma_1 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{(RA_{PP})\ \dfrac{}{\Gamma_1 \vdash [\texttt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}}}{(SCR_{EC})\ \dfrac{}{\Gamma_1 \vdash_T \texttt{g} :: \sigma(\alpha_1) = [[\alpha_5]]}}$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]], \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [[\alpha_5]] \to [[\alpha_5]], \alpha_4 \mapsto [\alpha_5]\}$ *is a solution of* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

We get $\Gamma_2 = \Gamma \cup \{\texttt{f} :: \forall a.[[a]], \texttt{g} :: \forall a.[[a]]\}$. *Since* $\Gamma_2 \neq \Gamma_1$, *another iteration is required. One can guess that this procedure does not end. We prove it: Let* $[a]^i$ *the i-fold nesting of lists. We use induction to show that* $\Gamma_i = \Gamma \cup \{\texttt{f} :: \forall a.[a]^i, \texttt{g} :: \forall a.[a]^i\}$. *The base is already shown. The induction step is*

$$\dfrac{\dfrac{\dfrac{(AxC)\ \overline{\Gamma_i \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}, \quad (AxSC)\ \overline{\Gamma_i \vdash \texttt{g} :: [\alpha_5]^i}}{(RAPP)\ \Gamma_i \vdash (\texttt{Cons g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3\}}, \quad (AxC)\ \overline{\Gamma_i \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{(RAPP)\ \Gamma_i \vdash [\texttt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}}{(SCREC)\ \Gamma_i \vdash_T \texttt{f} :: \sigma(\alpha_1) = [[\alpha_5]^i]}$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \to [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\}$ *is a solution of* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

$$\dfrac{\dfrac{\dfrac{(AxC)\ \overline{\Gamma_i \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}, \quad (AxSC)\ \overline{\Gamma_i \vdash \texttt{f} :: [\alpha_5]^i}}{(RAPP)\ \Gamma_i \vdash (\texttt{Cons f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3\}}, \quad (AxC)\ \overline{\Gamma_i \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{(RAPP)\ \Gamma_i \vdash [\texttt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}}{(SCREC)\ \Gamma_i \vdash_T \texttt{g} :: \sigma(\alpha_1) = [[\alpha_5]^i]}$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \to [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\}$ *is a solution of* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

We derive $\Gamma_{i+1} = \Gamma \cup \{\texttt{f} :: \forall a.[a]^{i+1}, \texttt{g} :: \forall a.[a]^{i+1}\}$.

Proposition 5.5.7. *The iterative type inference algorithm may not terminate.*

Proof. See Example 5.5.6. □

It can be shown, that the iterative typing in general is undecidable (and thus non termination of our algorithm is a consequence, and not a bad design of the algorithm).

Theorem 5.5.8. *Iterative typing is undecidable.*

Proof. We do not give the proof. It follows from the undecidability of so-called semi-unification of first-order terms. The proofs can be found in (Kfoury et al., 1990b; Kfoury et al., 1993; Henglein, 1993). □

Finally, we show that *type safety* holds for iterative typing. Type safety requires two properties: Typing is preserved when evaluating an expression (called *type preservation*) and evaluation of well-typed expressions does not get stuck until a value is obtained (called *progress lemma*).

Lemma 5.5.9. *Let s be a directly dynamically untyped* KFPTS+seq*-expression. Then the iterative typing cannot type s.*

Proof. If s is directly dynamically untyped, then one of the following cases holds:

- $s = R[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ Alts]$ and c is not of type T. If iterative typing tries to infer the type of the `case`-expression, equations are added to make sure that the type of $(c\ s_1\ \ldots\ s_n)$ is equal to the type of the patterns in the `case`-alternative. Since the types of the patterns belong to type T, but c does not, unification of the equations will fail.

- $s = R[\texttt{case}_T\ \lambda x.t\ \texttt{of}\ Alts]$: If the `case`-expression is typed, iterative typing will add equations that ensure that the type of $\lambda x.t$ is a function type and equal to the type of the patterns. Unification of these equation will fail, because the patterns are not of function type.
- $R[(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ t]$: the iterative typing types $((c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ t)$ as a nested application $(((c\ s_1)\ \ldots)\ s_{\mathrm{ar}(c)})\ t)$. For each application, equations are added that imply that c can receive at most $\mathrm{ar}(c)$ arguments. Since there is one more argument, unification will fail. □

Lemma 5.5.10 (Type Preservation)**.** *Let s be a well-typed and closed* KFPTSP+seq*-expression (of a well-typed* KFPTSP+seq*-program) and $s \xrightarrow{name} s'$. Then s' is well-typed.*

Proof. For the proof, all cases of a (β)-, (SC-β)- and (case)-reduction have to be inspected. For the type derivation of s, all type derivations of subterms of s can be read off. In the reduction, the types are moved and copied. It is easy to verify that a type derivation is still possible after the reduction. □

Considering the type, the last lemma can be formulated as: For a monomorphic type τ: If $t : \tau$ and $t \xrightarrow{name} t'$, then $t' : \tau$. Note that it may happen that after a reduction step the type is more general than before.

Lemmas 5.5.9 and 5.5.10 show:

Proposition 5.5.11. *Let s be a well-typed, closed* KFPTSP+seq*-expression. Then s is not dynamically untyped.*

Lemma 5.5.12 (Progress Lemma)**.** *Let s be a well-typed, closed* KFPTSP+seq*-expression. Then*

- *s is a WHNF, or*
- *s is call-by-name-reducible, i.e. $s \xrightarrow{name} s'$ for some s'.*

Proof. Consider the cases where a KFPTS+seq-expression is not a WHNF, closed, and irreducible. In all cases s is directly dynamically untyped and thus for a well-typed expression, these cases cannot occur. □

Theorem 5.5.13. *Type safety holds for the iterative typing of* KFPTSP+seq.

5.5.2.4. Forcing Termination

To force termination of the iterative type inference algorithm, a so-called Milner-step can be done, which will be explained below. In the next section we will introduce the Hindley-Damas-Milner type inference algorithm which also uses the Milner-step, but the difference is that it uses the Milner-step in the first iteration and so there are no iterations at all. Using the Milner-step within the iterative type inference algorithm, it may be possible to compute more general types, e.g. by perfoming 10 iterations and then stopping using the Milner-step.

Definition 5.5.14 (Milner-Step)**.** *Let $SC_1, \ldots, SC_m$ be a group of mutually recursive supercombinators- and let $\Gamma_i \vdash_T SC_1 :: \tau_1, \ldots, \Gamma_i \vdash_T SC_m :: \tau_m$ be the assumptions computed by the i-th iteration for $SC_1, \ldots, SC_m$.*

For the Milner-step: Type all supercombinators $SC_1, \ldots, SC_m$ together using the type assumption $\Gamma_M = \Gamma \cup \{SC_1 :: \tau_1, \ldots, SC_m :: \tau_m\}$ (where Γ contains the types of the constructors and other supercombinators that are already typed) and using the following rule:

$$\text{(SCRecMilnerStep)}\ \frac{\begin{array}{c}\text{for } i = 1, \ldots, m: \\ \Gamma_M \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau_i', E_i\end{array}}{\Gamma_M \vdash_T \text{for } i = 1, \ldots, m\ SC_i :: \sigma(\alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i')}$$

$$\text{if } \sigma \text{ is the solution of } E_1 \cup \ldots \cup E_m \cup \bigcup_{i=1}^{m} \{\tau_i \doteq \alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i'\}$$

$$\begin{array}{rcl} \text{and} \quad SC_1\ x_{1,1}\ \ldots\ x_{1,n_1} & = & s_1 \\ \ldots & & \\ SC_m\ x_{m,1}\ \ldots\ x_{m,n_m} & = & s_m \end{array}$$

$$\text{are the definitions of } SC_1, \ldots, SC_m$$

As a new inference rule in type derivation we add:

$$\text{(AxSC2)}\ \frac{}{\Gamma \cup \{SC :: \tau\} \vdash SC :: \tau}$$

if τ is not universally quantified

We explain the Milner-step: In the assumption Γ_M the types of the supercombinator to be typed are *not* universally quantified. The rule (AxSC2) is then used to extract such a type from the assumptions. No renaming is performed, i.e. the whole typing uses one fixed type for each occurrence of the supercombinator. Additionally, in the rule (SCRecMilnerStep) all equations are unified together and new equations are added: For each supercombinator the assumed type must match the inferred types.

As consequence, the new assumption is always consistent, and thus no further iteration is required.

Note that in unlike (SCRec) all supercombinators that are mutually recursive must be treated together to ensure that unification considers all equations together.

A disadvantage of the algorithm is that one does not know, when to use the Milner-step and that the types are more restrictive than iterative types. It may happen that a further iteration would lead to a type, but the Milner step rejects the expression as untyped.

5.5.3. Hindley-Damas-Milner-Typing

The Hindley-Damas-Milner typing was invented and analyzed by Robin Milner, Luis Damas and Roger Hindley. Instead of iterative typing, the algorithm uses a Milner-step as first step and thus it terminates after this step:

Algorithm 4: Hindley-Damas-Milner Type Inference Algorithm

Let $SC_1, \ldots, SC_m$ be all mutually recursive supercombinators of an equivalence class w.r.t. $\simeq$ and assume that the supercombinators strictly less than $SC_1, \ldots, SC_m$ w.r.t. $\preceq$ are already typed.

1. The type assumption Γ contains the already typed supercombinators with universally quantified types and also the types of the constructors.

2. Type $SC_1, \ldots, SC_m$ with rule (HDMSCRec):

$$\text{(HDMSCRec)}\ \frac{\begin{array}{c}\text{for } i = 1, \ldots, m: \\ \Gamma \cup \{SC_1 :: \beta_1, \ldots, SC_m :: \beta_m\} \\ \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau_i, E_i\end{array}}{\Gamma \vdash_T \text{for } i = 1, \ldots, m\ SC_i :: \sigma(\alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i)}$$

if σ is the solution of $E_1 \cup \ldots \cup E_m \cup \bigcup_{i=1}^{m} \{\beta_i \doteq \alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i\}$

and the definitions of $SC_1, \ldots, SC_m$ are

$$\begin{array}{rcl} SC_1\ x_{1,1}\ \ldots\ x_{1,n_1} & = & s_1 \\ \ldots & & \\ SC_m\ x_{m,1}\ \ldots\ x_{m,n_m} & = & s_m \end{array}$$

If the unification fails, then $SC_1, \ldots, SC_m$ are not Hindley-Damas-Milner-typeable, otherwise the Hindley-Damas-Milner-types of $SC_1, \ldots, SC_m$ are the types derived by rule (HDMSCRec). Note that rule (AxSC2) is used in the type derivation.

To explain the algorithm, we also write down the rule (HDMSCRec) for the case of a single recursive supercombinator:

$$\text{(HDMSCRec1)}\ \frac{\Gamma \cup \{SC :: \beta, x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{\Gamma \vdash_T SC :: \sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau)}$$

if σ is the solution of $E \cup \{\beta \doteq \alpha_1 \to \ldots \to \alpha_n \to \tau\}$

and $SC\ x_1\ \ldots\ x_n = s$ is the definition of SC

We explain this rule: the type assumption for the supercombinator is β, so the most general type. In difference to the iterative type inference it is not universally quantified and thus when typing the right-hand side s of the definition, all occurrences of SC are typed with the same type β (and not with renamed copies). (iterative type inference uses rule (AxSC), Hindley-Damas-Milner type inference uses (AxSC2)). Another difference to the iterative type inference is the additional unification equation $\beta \doteq \alpha_1 \to \ldots \to \alpha_n \to \tau$. It forces the assumed type β to be equal to the inferred type.

The Hindley-Damas-Milner type inference algorithm has the following properties:

- It terminates.
- It returns unique types (up to renaming of variables).
- Hindley-Damas-Milner typing is decidable.

- The decidable problem whether an expression is Hindley-Damas-Milner typeable is DEXPTIME-complete (see (Mairson, 1990; Kfoury et al., 1990a)).
- It may return more restrictive types than iterative type inference. In particular, there are expressions that are iteratively typeable, but not Hindley-Damas-Milner typeable.

Example 5.5.15. *An example that requires exponentially many type variables is:*

$$\begin{array}{lcl} x_0 & = & \lambda z.z \\ x_1 & = & (x_0, x_0) \\ x_2 & = & (x_1, x_1) \\ x_3 & = & (x_2, x_2) \\ & \ldots & \\ x_n & = & (x_{n-1}, x_{n-1}) \end{array}$$

The type of x_n contains 2^n type variables.

We consider some examples.

Example 5.5.16. *We type the supercombinator* `map` *with the Hindley-Damas-Milner type inference algorithm.*

```
map f xs = case xs of {
             [] -> []
             (y:ys) -> (f y):(map f ys)
           }
```

The assumption for the constructors is $\Gamma_0 = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} :: \forall a.[a]\}$.
Let $\Gamma = \Gamma_0 \cup \{\texttt{map} :: \beta, f :: \alpha_1, xs :: \alpha_2\}$ *and* $\Gamma' = \Gamma' \cup \{y : \alpha_3, ys :: \alpha_4\}$.
Typing starts with rule (HDMSCRec):

$$\text{(HDMSCRec1)}\ \frac{\text{(RCase)}\ \dfrac{\begin{array}{ll}(a) & \Gamma' \vdash xs :: \tau_1, E_1 \\ (b) & \Gamma' \vdash \texttt{Nil} :: \tau_2, E_2 \\ (c) & \Gamma' \vdash (\texttt{Cons}\ y\ ys) :: \tau_3, E_3 \\ (d) & \Gamma' \vdash \texttt{Nil} :: \tau_4, E_4 \\ (e) & \Gamma' \vdash (\texttt{Cons}\ (f\ y)\ (\texttt{map}\ f\ ys)) :: \tau_5, E_5 \end{array}}{\Gamma' \vdash \texttt{case}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; \texttt{Cons}\ y\ ys \to \texttt{Cons}\ y\ (\texttt{map}\ f\ ys)\} :: \alpha, E}}{\Gamma \vdash_T \texttt{map} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha)}$$

if σ is the solution of $E \cup \{\beta \doteq \alpha_1 \to \alpha_2 \to \alpha\}$

where $E = E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \doteq \tau_2, \tau_1 \doteq \tau_3, \alpha \doteq \tau_4, \alpha \doteq \tau_5\}$. *Types* $\tau_1, \ldots, \tau_5$ *and equations* $E_1, \ldots, E_5$ *are computed during deriving the premises* $(a), \ldots, (e)$.
We write down the derivations separately:

(a) $\dfrac{}{\Gamma' \vdash xs :: \alpha_2, \emptyset}\,(AxV)$

I.e., $\tau_1 = \alpha_2$ *and* $E_1 = \emptyset$.

(b) $\dfrac{}{\Gamma' \vdash \texttt{Nil} :: [\alpha_5], \emptyset}\,(AxC)$

I.e., $\tau_2 = [\alpha_5]$ *and* $E_2 = \emptyset$

(c)
$$\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \texttt{Cons} :: \alpha_6 \to [\alpha_6] \to [\alpha_6], \emptyset}\,(AxC) \quad \dfrac{}{\Gamma' \vdash y :: \alpha_3, \emptyset}\,(AxV)}{\Gamma' \vdash (\texttt{Cons}\ y) :: \alpha_7, \{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7\}}\,(RA_{PP}) \quad \dfrac{}{\Gamma' \vdash ys :: \alpha_4, \emptyset}\,(AxV)}{\Gamma' \vdash (\texttt{Cons}\ y\ ys) :: \alpha_8, \{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7, \alpha_7 \doteq \alpha_4 \to \alpha_8\}}\,(RA_{PP})$$

I.e., $\tau_3 = \alpha_8$ *and* $E_3 = \{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7, \alpha_7 \doteq \alpha_4 \to \alpha_8\}$

(d) $\dfrac{}{\Gamma' \vdash \texttt{Nil} :: [\alpha_9], \emptyset}\,(AxC)$

I.e., $\tau_4 = [\alpha_9]$ *and* $E_4 = \emptyset$.

(e)
$$\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \texttt{Cons} :: \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}], \emptyset}\,(AxC) \quad \dfrac{\dfrac{}{\Gamma' \vdash f :: \alpha_1, \emptyset}\,(AxV) \quad \dfrac{}{\Gamma' \vdash y :: \alpha_3, \emptyset}\,(AxV)}{\Gamma' \vdash (f\ y) :: \alpha_{15}, \{\alpha_1 \doteq \alpha_3 \to \alpha_{15}\}}\,(RA_{PP})}{\Gamma' \vdash (\texttt{Cons}\ (f\ y)) :: \alpha_{11}, \{\alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}\}}\,(RA_{PP}) \quad \dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \texttt{map} :: \beta, \emptyset}\,(AxSC2) \quad \dfrac{}{\Gamma' \vdash f :: \alpha_1, \emptyset}\,(AxV)}{\Gamma' \vdash (\texttt{map}\ f) :: \alpha_{12}, \{\beta \doteq \alpha_1 \to \alpha_{12}\}}\,(RA_{PP}) \quad \dfrac{}{\Gamma' \vdash ys :: \alpha_4, \emptyset}\,(AxV)}{\Gamma' \vdash (\texttt{map}\ f\ ys) :: \alpha_{13}, \{\beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}}\,(RA_{PP})}{\Gamma' \vdash (\texttt{Cons}\ (f\ y)\ (\texttt{map}\ f\ ys)) :: \alpha_{14}, \{\alpha_{11} \doteq \alpha_{13} \to \alpha_{14}, \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}, \beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}}\,(RA_{PP})$$

I.e., $\tau_5 = \alpha_{14}$ *and*
$E_5 = \{\alpha_{11} \doteq \alpha_{13} \to \alpha_{14}, \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15},$
$\beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}$

In total, the equations $E \cup \{\beta \doteq \alpha_1 \to \alpha_2 \to \alpha\}$ *have to be unified, i.e.*

$$\begin{aligned}&\{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7, \alpha_7 \doteq \alpha_4 \to \alpha_8, \alpha_{11} \doteq \alpha_{13} \to \alpha_{14},\\ &\ \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}, \beta \doteq \alpha_1 \to \alpha_{12},\\ &\ \alpha_{12} \doteq \alpha_4 \to \alpha_{13}, \alpha_2 \doteq [\alpha_5], \alpha_2 \doteq \alpha_8, \alpha \doteq \alpha_9, \alpha \doteq \alpha_{14}\}\end{aligned}$$

Unification results in

$$\begin{aligned}\sigma = \{&\alpha \mapsto [\alpha_{10}], \alpha_1 \mapsto \alpha_6 \to \alpha_{10}, \alpha_2 \mapsto [\alpha_6], \alpha_3 \mapsto \alpha_6, \alpha_4 \mapsto [\alpha_6], \alpha_5 \mapsto \alpha_6,\\ &\alpha_7 \mapsto [\alpha_6] \to [\alpha_6], \alpha_8 \mapsto [\alpha_6], \alpha_9 \mapsto [\alpha_{10}], \alpha_{11} \mapsto [\alpha_{10}] \to [\alpha_{10}],\\ &\alpha_{12} \mapsto [\alpha_6] \to [\alpha_{10}], \alpha_{13} \mapsto [\alpha_{10}], \alpha_{14} \mapsto [\alpha_{10}], \alpha_{15} \mapsto \alpha_{10},\\ &\beta \mapsto (\alpha_6 \to \alpha_{10}) \to [\alpha_6] \to [\alpha_{10}],\end{aligned}$$

I.e., $map :: \sigma(\alpha_1 \to \alpha_2 \to \alpha) = (\alpha_6 \to \alpha_{10}) \to [\alpha_6] \to [\alpha_{10}]$.

Example 5.5.17. *We consider the Hindley-Damas-Milner typing of Example 5.5.3. Let super-*

combinator g *be defined as*

```
g x = x : (g (g 'c'))
```

The assumption for the constructors is $\Gamma = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{'c'} :: \texttt{Char}\}$. *Let* $\Gamma' = \Gamma \cup \{x :: \alpha, \texttt{g} :: \beta\}$.

$$\dfrac{\dfrac{\dfrac{\overset{(AxC)}{\Gamma' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset} \quad \dfrac{\overset{(AxV)}{\Gamma' \vdash x :: \alpha, \emptyset}}{\overset{(RA_{PP})}{\Gamma' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \dot{=} \alpha \to \alpha_3}}}{\overset{(RA_{PP})}{\Gamma' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\beta \dot{=} \texttt{Char} \to \alpha_7, \beta \dot{=} \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \dot{=} \alpha \to \alpha_3, \alpha_3 \dot{=} \alpha_4 \to \alpha_2\}}} \quad \dfrac{\dfrac{\overset{(AxSC2)}{\Gamma' \vdash \texttt{g} :: \beta, \emptyset} \quad \dfrac{\overset{(AxC)}{\Gamma' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}}{\overset{(RA_{PP})}{\Gamma' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\beta \dot{=} \texttt{Char} \to \alpha_7\}}}}{\overset{(RA_{PP})}{\Gamma' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\beta \dot{=} \texttt{Char} \to \alpha_7, \beta \dot{=} \alpha_7 \to \alpha_4\}}}}{\overset{(HDMSCR_{EC})}{\Gamma \vdash_T \texttt{g} :: \sigma(\alpha \to \alpha_2)}}$$

where σ *is the solution of* $\{\beta \dot{=} \texttt{Char} \to \alpha_7, \beta \dot{=} \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \dot{=} \alpha \to \alpha_3, \alpha_3 \dot{=} \alpha_4 \to \alpha_2, \beta \dot{=} \alpha \to \alpha_2\}$

Unification fails, because Char *has to be unified with a list. Hence,* g *is not Hindley-Damas-Milner typeable.*

Example 5.5.18. *We consider the supercombinator* g *of Example 5.5.2:*

```
g x = 1 : (g (g 'c'))
```

Let $\Gamma' = \Gamma \cup \{x :: \alpha, \texttt{g} :: \beta\}$.

$$\dfrac{\dfrac{\dfrac{\overset{(AxC)}{\Gamma' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset} \quad \dfrac{\overset{(AxC)}{\Gamma' \vdash 1 :: \texttt{Int}, \emptyset}}{\overset{(RA_{PP})}{\Gamma' \vdash (\texttt{Cons}\ 1) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \dot{=} \texttt{Int} \to \alpha_3}}}{\overset{(RA_{PP})}{\Gamma' \vdash \texttt{Cons}\ 1\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\beta \dot{=} \texttt{Char} \to \alpha_7, \beta \dot{=} \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \dot{=} \texttt{Int} \to \alpha_3, \alpha_3 \dot{=} \alpha_4 \to \alpha_2\}}} \quad \dfrac{\dfrac{\overset{(AxSC2)}{\Gamma' \vdash \texttt{g} :: \beta, \emptyset} \quad \dfrac{\overset{(AxC)}{\Gamma' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}}{\overset{(RA_{PP})}{\Gamma' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\beta \dot{=} \texttt{Char} \to \alpha_7\}}}}{\overset{(RA_{PP})}{\Gamma' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\beta \dot{=} \texttt{Char} \to \alpha_7, \beta \dot{=} \alpha_7 \to \alpha_4\}}}}{\overset{(SCR_{EC})}{\Gamma \vdash_T \texttt{g} :: \sigma(\alpha \to \alpha_2)}}$$

where σ *is the solution of* $\{\beta \dot{=} \texttt{Char} \to \alpha_7, \beta \dot{=} \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \dot{=} \texttt{Int} \to \alpha_3, \alpha_3 \dot{=} \alpha_4 \to \alpha_2, \beta \dot{=} \alpha \to \alpha_2\}$

Unification fails, due to the equation $[\alpha_5] \dot{=} \texttt{Char}$.

Finally, we consider a supercombinator that has a type w.r.t. the Hindley-Damas-Milner type inference and w.r.t. the iterative type inference, but the iterative type is more general.

Example 5.5.19. *Assume the recursive data type* Tree*, defined in Haskell as*

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

The types of the constructors are: Empty $:: \forall a.$Tree a *and* Node $:: \forall a.a \to$ *Tree a* $\to$ *Tree a* $\to$ *Tree a*

We type the supercombinator g *defined as*

```
g x y = Node True (g x y) (g y x)
```

We first compute the Hindley-Damas-Milner type: Suppose that Γ *contains the assumption on the constructors. Let* $\Gamma' = \Gamma \cup \{x :: \alpha_1, y :: \alpha_2, \mathtt{g} :: \beta\}$*. For readability, we draw the derivation trees separately:*

$$
\textit{(HDMSCR}_{\textit{EC}}\textit{)}\ \frac{\textit{(RA}_{\textit{PP}}\textit{)}\ \dfrac{(c),(a)}{\begin{array}{c}\Gamma' \vdash (\mathtt{Node\ True}\ (\mathtt{g}\ x\ y)\ (\mathtt{g}\ y\ x)) :: \alpha_3,\\ \{\alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \doteq \mathtt{Bool} \to \alpha_9,\\ \beta \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5, \alpha_9 \doteq \alpha_7 \to \alpha_{10},\\ \beta \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7, \alpha_{10} \doteq \alpha_5 \to \alpha_3\}\end{array}}}{\begin{array}{c}\Gamma \vdash_T \mathtt{g} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3)\\ \textit{where } \sigma \textit{ is the solution of}\\ \{\alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \doteq \mathtt{Bool} \to \alpha_9,\\ \beta \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5, \alpha_9 \doteq \alpha_7 \to \alpha_{10},\\ \beta \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7, \alpha_{10} \doteq \alpha_5 \to \alpha_3, \beta \doteq \alpha_1 \to \alpha_2 \to \alpha_3\}\end{array}}
$$

(a)

$$
\textit{(RA}_{\textit{PP}}\textit{)}\ \frac{\textit{(RA}_{\textit{PP}}\textit{)}\ \dfrac{\textit{(AxSC2)}\ \dfrac{}{\Gamma' \vdash g :: \beta, \emptyset}, \textit{(AxV)}\ \dfrac{}{\Gamma' \vdash y :: \alpha_2, \emptyset}}{\Gamma' \vdash (\mathtt{g}\ y) :: \alpha_4, \{\beta \doteq \alpha_2 \to \alpha_4\}}, \textit{(AxV)}\ \dfrac{}{\Gamma' \vdash x :: \alpha_1, \emptyset}}{(\mathtt{g}\ y\ x)) :: \alpha_5, \{\beta \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5\}}
$$

(b)

$$
\textit{(RA}_{\textit{PP}}\textit{)}\ \frac{\textit{(RA}_{\textit{PP}}\textit{)}\ \dfrac{\textit{(AxSC2)}\ \dfrac{}{\Gamma' \vdash g :: \beta, \emptyset}, \textit{(AxV)}\ \dfrac{}{\Gamma' \vdash x :: \alpha_1, \emptyset}}{\Gamma' \vdash (\mathtt{g}\ x) :: \alpha_6, \{\beta \doteq \alpha_1 \to \alpha_6\}}, \textit{(AxV)}\ \dfrac{}{\Gamma' \vdash y :: \alpha_2, \emptyset}}{(\mathtt{g}\ x\ y)) :: \alpha_7, \{\beta \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7\}}
$$

(c)

$$
\textit{(RA}_{\textit{PP}}\textit{)}\ \frac{\textit{(RA}_{\textit{PP}}\textit{)}\ \dfrac{\textit{(AxC)}\ \dfrac{}{\Gamma' \vdash \mathtt{Node} :: \alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8, \emptyset}, \textit{(AxC)}\ \dfrac{}{\Gamma' \vdash \mathtt{True} :: \mathtt{Bool}, \emptyset}}{\begin{array}{c}\Gamma' \vdash (\mathtt{Node\ True}) :: \alpha_9,\\ \{\alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \doteq \mathtt{Bool} \to \alpha_9\}\end{array}}, (b)}{\begin{array}{c}\Gamma' \vdash (\mathtt{Node\ True}\ (\mathtt{g}\ x\ y)) :: \alpha_{10},\\ \{\alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \doteq \mathtt{Bool} \to \alpha_9,\\ \alpha_9 \doteq \alpha_7 \to \alpha_{10}, \beta \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7\}\end{array}}
$$

Unification results in

$$\begin{aligned}\sigma \quad = \{ \quad & \alpha_1 \mapsto \alpha_2, \\ & \alpha_3 \mapsto \texttt{Tree Bool}, \\ & \alpha_4 \mapsto \alpha_2 \to \texttt{Tree Bool}, \\ & \alpha_5 \mapsto \texttt{Tree Bool}, \\ & \alpha_6 \mapsto \alpha_2 \to \texttt{Tree Bool}, \\ & \alpha_7 \mapsto \texttt{Tree Bool}, \\ & \alpha_8 \mapsto \texttt{Bool}, \\ & \alpha_9 \mapsto \texttt{Tree Bool} \to \texttt{Tree Bool} \to \texttt{Tree Bool}, \\ & \alpha_{10} \mapsto \texttt{Tree Bool} \to \texttt{Tree Bool}, \\ & \beta \mapsto \alpha_2 \to \alpha_2 \to \texttt{Tree Bool} \quad \}\end{aligned}$$

I.e., $\Gamma \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3) = \alpha_2 \to \alpha_2 \to \texttt{Tree Bool}$.

The Hindley-Damas-Milner type inference algorithm thus results in $\texttt{g} :: a \to a \to \texttt{Tree Bool}$.

We now consider the iterative type inference for g*: Let* $\Gamma_0 = \Gamma \cup \{\texttt{g} :: \forall\alpha.\alpha\}$. *The first iteration for* Γ_0 *consists of the following derivation (where* $\Gamma_0' = \Gamma_0 \cup \{x :: \alpha_1, y :: \alpha_2\}$*):*

$$\textit{(SCRec)}\ \frac{\textit{(RApp)}\ \dfrac{(c), (a)}{\begin{array}{c}\Gamma_0' \vdash (\texttt{Node True } (\texttt{g } x\ y)\ (\texttt{g } y\ x)) :: \alpha_3, \\ \{\alpha_8 \to \texttt{Tree } \alpha_8 \to \texttt{Tree } \alpha_8 \to \texttt{Tree } \alpha_8 \doteq \texttt{Bool} \to \alpha_9, \\ \beta_1 \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5, \alpha_9 \doteq \alpha_7 \to \alpha_{10}, \\ \beta_2 \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7, \alpha_{10} \doteq \alpha_5 \to \alpha_3\}\end{array}}}{\begin{array}{c}\Gamma_0 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3) \\ \textit{where } \sigma \textit{ is the solution of} \\ \{\alpha_8 \to \texttt{Tree } \alpha_8 \to \texttt{Tree } \alpha_8 \to \texttt{Tree } \alpha_8 \doteq \texttt{Bool} \to \alpha_9, \\ \beta_1 \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5, \alpha_9 \doteq \alpha_7 \to \alpha_{10}, \\ \beta_2 \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7, \alpha_{10} \doteq \alpha_5 \to \alpha_3\}\end{array}}$$

(a)

$$\textit{(RApp)}\ \frac{\textit{(RApp)}\ \dfrac{\textit{(AxSC)}\ \dfrac{}{\Gamma_0' \vdash g :: \beta_1, \emptyset}, \textit{(AxV)}\ \dfrac{}{\Gamma_0' \vdash y :: \alpha_2, \emptyset}}{\Gamma_0' \vdash (\texttt{g } y) :: \alpha_4, \{\beta_1 \doteq \alpha_2 \to \alpha_4\}}, \textit{(AxV)}\ \dfrac{}{\Gamma_0' \vdash x :: \alpha_1, \emptyset}}{(\texttt{g } y\ x)) :: \alpha_5, \{\beta_1 \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5\}}$$

(b)

$$\dfrac{\dfrac{(AxSC)\ \dfrac{}{\Gamma_0' \vdash g :: \beta_2, \emptyset}, \quad (AxV)\ \dfrac{}{\Gamma_0' \vdash x :: \alpha_1, \emptyset}}{(RAPP)\ \Gamma_0' \vdash (\mathtt{g}\ x) :: \alpha_6, \{\beta_2 \doteq \alpha_1 \to \alpha_6\}} \quad , \quad (AxV)\ \dfrac{}{\Gamma_0' \vdash y :: \alpha_2, \emptyset}}{(RAPP)\ (\mathtt{g}\ x\ y)) :: \alpha_7, \{\beta_2 \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7\}}$$

(c)

$$\dfrac{\dfrac{(AxC)\ \dfrac{}{\Gamma_0' \vdash \mathtt{Node} :: \alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8, \emptyset}, \quad (AxC)\ \dfrac{}{\Gamma_0' \vdash \mathtt{True} :: \mathtt{Bool}, \emptyset}}{(RAPP)\ \begin{array}{c}\Gamma_0' \vdash (\mathtt{Node}\ \mathtt{True}) :: \alpha_9, \\ \{\alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \doteq \mathtt{Bool} \to \alpha_9\}\end{array}} \quad , \quad (b)}{(RAPP)\ \begin{array}{c}\Gamma_0' \vdash (\mathtt{Node}\ \mathtt{True}\ (\mathtt{g}\ x\ y)) :: \alpha_{10}, \\ \{\alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \to \mathtt{Tree}\ \alpha_8 \doteq \mathtt{Bool} \to \alpha_9, \\ \alpha_9 \doteq \alpha_7 \to \alpha_{10}, \beta_2 \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7\}\end{array}}$$

Unification results in

$$\begin{array}{rl}\sigma = \{ & \beta_1 \mapsto \alpha_2 \to \alpha_1 \to \mathtt{Tree}\ \mathtt{Bool}, \\ & \beta_2 \mapsto \alpha_1 \to \alpha_2 \to \mathtt{Tree}\ \mathtt{Bool}, \\ & \alpha_3 \mapsto \mathtt{Tree}\ \mathtt{Bool}, \\ & \alpha_4 \mapsto \alpha_1 \to \mathtt{Tree}\ \mathtt{Bool}, \\ & \alpha_5 \mapsto \mathtt{Tree}\ \mathtt{Bool}, \\ & \alpha_6 \mapsto \alpha_2 \to \mathtt{Tree}\ \mathtt{Bool}, \\ & \alpha_7 \mapsto \mathtt{Tree}\ \mathtt{Bool}, \\ & \alpha_8 \mapsto \mathtt{Bool}, \\ & \alpha_9 \mapsto \mathtt{Tree}\ \mathtt{Bool} \to \mathtt{Tree}\ \mathtt{Bool} \to \mathtt{Tree}\ \mathtt{Bool}, \\ & \alpha_{10} \mapsto \mathtt{Tree}\ \mathtt{Bool} \to \mathtt{Tree}\ \mathtt{Bool} \quad \}\end{array}$$

This results in $\mathtt{g} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3) = \alpha_1 \to \alpha_2 \to \mathtt{Tree}\ \mathtt{Bool}$. *Now* $\Gamma_1 = \Gamma \cup \{\mathtt{g} :: \forall \alpha, \beta, \alpha \to \beta \to \mathtt{Tree}\ \mathtt{Bool}\}$. *A further iteration shows that* Γ_1 *is consistent (we omit it)*

Thus the iterative type inference algorithm returns $\mathtt{g} :: a \to b \to \mathtt{Tree\ Bool}$ *and the iterative type is more general than the Hindley-Damas-Milner-Type. In the Haskell interpreter, the more specific type is derived:*

```
*Main> let g x y =  Node True (g x y) (g y x)
*Main> :t g
g :: t -> t -> Tree Bool
```

If the programmer assigns the more general type, then Haskell verifies it:

```
*Main> let g :: a -> b -> Tree Bool; g x y =  Node True (g x y) (g y x)
*Main> :t g
g :: a -> b -> Tree Bool
```

The Hindley-Damas-Milner type inference algorithm has the property, that well-typed programs are never dynamically untyped. Also, the progress lemma holds, i.e. Hindley-Damas-Milner typed programs are reducible or WHNFs. Type preservation also holds n KFPTSP+seq for Hindley-Damas-Milner typing. If recursive `let`-expressions are used instead of supercombinators, then it may happen that Hindley-Damas-Milner typed expressions are reduced to expressions that are no longer Hindley-Damas-Milner typed (nevertheless the expression is still iteratively typed, so no dynamic type error will occur) An example is

```
let x = (let y = \u -> z in (y [], y True, seq x True)); z = const z x in x
```

The expression is Hindley-Damas-Milner typeable. After a so-called $(llet)$-reduction which flattens the let-bindings, the expression becomes

```
let x = (y [], y True, seq x True); y = \u -> z; z = const z x in x
```

This expression is no longer Hindley-Damas-Milner typeable, since `y` and `x` are typed together.

5.6. Conclusion and References

We introduced the polymorphic type systems for KFPTSP+seq-expressions and supercombinators, where we explained two type inference algorithms and analyzed their properties. The content of this chapter mainly stems from (Schmidt-Schauß, 2009).

The Hindley-Damas-Milner type inference algorithm was invented in (Milner, 1978) and similarly already in (Hindley, 1969). In (Damas & Milner, 1982) soundness and completeness of the algorithm was shown. The iterative type inference algorithm mainly stems from (Mycroft, 1984).

6. Semantics

6.1. Formal Semantics

A semantics of a programming language is used to formally describe the behavior of programs and its effect on the environment. Thus it represents the meaning of a programming. Such a semantics is necessary to reason on correctness of program optimization, program translation, program transformation and for verifying programs.

The research field of *formal semantics for programming languages* (see e.g. (Gunter, 1992; Winskel, 1993; Mitchell, 1996; Stump, 2013)) investigates this topic. In general there exist the following approaches of semantics for programming languages which can be briefly described as follows:

An *axiomatic semantics* defines the meaning of programs using logical axioms. Properties of programs can then be deduced using logical inference rules. A prominent example for an axiomatic semantics is the Hoare calculus (see (Hoare, 1969)) where triples $\{P\}\ C\ \{Q\}$) are used to describe the effect of the programs on the environment: if precondition P holds and command C is executed, then postcondition Q holds. An inference rule is the following rule for sequencing:

$$\frac{\{P\}\ C_1\ \{Q\}, \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1; C_2\ \{R\}}$$

An *operational semantics* defines *how* a program is executed, i.e. it describes the evaluation of programs. There are different models used for operational semantics: *state transition systems* where a state describes the current state of the program and maybe the machine memory. The transitions define how the next state is computed. Using *abstract machines* a machine model is used to define the execution of programs. Finally, *rewriting systems* define how programs are rewritten to compute the successor in a sequence of an evaluation. We already used rewriting systems to describe the operational semantics of the lambda calculus and the functional core languages. The Turing machine can also be seen as an abstract machine, however the programs are hard coded in the set of states, thus there the program is not adjustable. In contrast, a universal Turing machine interprets its input as a Turing machine, so it is an interpreter of Turing machine. This is an operational semantics, however the programming language (consisting of TM descriptions) is a bit unusual for a programmer.

A further criterion for classifying operational semantics is the distinction into *small-step* and *big-step* semantics. While the former describes evaluation using small steps, i.e. in general an evaluation consists of multiple steps, the latter describes the evaluation within few (mostly one)

steps. The operational semantics that we have defined for the lambda calculus and the functional core languages are small-step semantics.

Usually operational semantics are intuitive, but reasoning using operational semantics is often seen as difficult and complex.

A *denotational semantics* (see e.g. (Stoy, 1977; Schmidt, 1986; Tennent, 1994)) maps programs into mathematical objects which are used as the denotation of a program and thus represent the meaning of a program. A wide-spread approach for denotational semantics is using domains, i.e. partially ordered sets.

A *contextual semantics* defines contextual equivalence as an equality notion on programs. The equivalence class can then be seen as the semantics of the program. We already have seen the contextual semantics for the call-by-name and the call-by-value lambda calculus.

A *transformational semantics* usually transforms the program in a program of another language (or a sublanguage) and then the semantics of the target language is used. This can for instance be used to remove syntactic sugar or for example, when expressing recursive supercombinators with the fixpoint operator.

A desired property of every semantic description is *compositionality* which means that the semantics of a program can be computed from the semantics of the subprograms. For example, if $\langle\cdot\rangle$ compute the semantics, then $\langle s + t\rangle = \langle s\rangle + \langle t\rangle$ should hold.

In this chapter we will consider different forms of operational semantics and denotational semantics. We do not consider axiomatic semantics. Instead of a functional language, we consider a core language of imperative programming. The language is called IMP. It can for instance, be found in (Winskel, 1993).

Definition 6.1.1. *Let V be a non-terminal for storage locations (generating a countably infinite set Loc of storage locations $x, y, \ldots$). Let n, m represent arbitrary integers. The syntax of* IMP-programs *is generated by the non-terminal* **Cmd***, which uses* arithmetic expressions *generated by the non-terminal* **AExp** *and* boolean expressions *generated by the non-terminal* **BExp**

$$\mathbf{AExp} ::= n \mid V \mid \mathbf{AExp} + \mathbf{AExp} \mid \mathbf{AExp} - \mathbf{AExp} \mid \mathbf{AExp} * \mathbf{AExp}$$

$$\begin{aligned}\mathbf{BExp} ::= {} & \texttt{True} \mid \texttt{False} \mid \mathbf{AExp} = \mathbf{AExp} \mid \mathbf{AExp} \leq \mathbf{AExp} \\ & \mid \neg\mathbf{BExp} \mid \mathbf{BExp} \vee \mathbf{BExp} \mid \mathbf{Bexp} \wedge \mathbf{BExp}\end{aligned}$$

$$\begin{aligned}\mathbf{Cmd} ::= {} & \texttt{skip} \mid V := \mathbf{AExp} \mid \mathbf{Cmd}; \mathbf{Cmd} \\ & \mid \texttt{if}\ \mathbf{BExp}\ \texttt{then}\ \mathbf{Cmd}\ \texttt{else}\ \mathbf{Cmd}\ \texttt{fi} \mid \texttt{while}\ \mathbf{BExp}\ \texttt{do}\ \mathbf{Cmd}\ \texttt{od}\end{aligned}$$

Example 6.1.2. *We show some example programs.*

The program

$$y := 2; z := 4; x := y + z$$

will assign 2 to storage location y, 4 *to storage location z, and* 6 *to storage location x.*

The program

$$x := 1; y := 100; \texttt{while}\ 1 \leq y\ \texttt{do}\ x := x * y; y := y - 1\ \texttt{od}$$

computes $100!$.

The program

$$s := 0; i := 100; \texttt{while}\ 1 \leq i\ \texttt{do}\ s := s + i * i; i := i - 1\ \texttt{od}$$

computes the sum $\sum_{i=0}^{100} i^2$.

6.2. Operational Semantics

A *state* is a partial function $\sigma : Loc \to \mathbb{Z}$ such that $Dom(\sigma)$ is finite. I.e., it assigns an integer to a finite set of storage locations. Storage locations store numbers, but no boolean values.

The behavior of programs that access storage locations before initializing them, can be defined in different ways (for example, all locations could be initialized with a default value like 0). We treat such an access as a runtime error.

6.2.1. A Big-Step Semantics

We define the evaluation of arithmetic expressions, boolean expressions and commands as a big-step semantics. We introduce the notation and later axioms and inference rules.

Definition 6.2.1 (Evaluation Relation)**.** *Let* Σ *be the set of all states, i.e.*

$$\Sigma = \{\sigma \mid \sigma : Loc \to \mathbb{Z} \wedge Dom(\sigma)\ \textit{is finite}\}.$$

For $\sigma \in \Sigma$ *and* $x \in Loc$, $\sigma(x) \in \mathbb{Z}$ *is the value of storage location x, or if* σ *is not defined for x,* $\sigma(x) = \bot$.

A configuration $\langle s, \sigma\rangle$ *consists of a command, arithmetic expression, or boolean expression s and a state* $\sigma \in \Sigma$.

We use the evaluation relation $\downarrow$ *for all three kinds of configurations:*

- *For an arithmetic expression a, we write* $\langle a, \sigma\rangle \downarrow n$ *to denote that a evaluates to a number* $n \in \mathbb{Z}$ *in state* σ.
- *For a boolean expression b, we write* $\langle b, \sigma\rangle \downarrow$ `True` *or* $\langle b, \sigma\rangle \downarrow$ `False`*, to denote that b evaluates to boolean value* `True` *(or* `False`*, resp.) in state* σ.
- *For a command c, we write* $\langle c, \sigma\rangle \downarrow \sigma'$ *if c changes the state* σ *to state* σ'.

Note that $\downarrow$ is a relation and not necessarily a function. If we would define it as a function, we would enforce that evaluation is deterministic by definition, but this determinism should be a consequence of the definition. Or from another view, if we define it to be a function, but an evaluation rule is non-deterministic, then we (accidentally) introduce a wrong rule. Thus, allowing a relation, and then proving that it is in fact a function, seems to be the right way. However, there are cases where a relation is required, for instance, if our programming language would have a true random number generator.

In the following section we list axioms and derivation rules for the big-step semantics of IMP. We use the notation

$$\frac{\text{premises}}{\text{conclusion}}$$

6.2.1.1. Rules for Evaluation of Arithmetic Expressions

The rules for evaluation of arithmetic expressions are:

$$\text{(AxNum)}\ \frac{}{\langle n, \sigma\rangle \downarrow n} \qquad \text{(Sum)}\ \frac{\langle a_1, \sigma\rangle \downarrow n_1 \quad \langle a_2, \sigma\rangle \downarrow n_2}{\langle a_1 + a_2, \sigma\rangle \downarrow n'} \text{ if } n' = n_1 + n_2$$

$$\text{(AxLoc)}\ \frac{}{\langle x, \sigma\rangle \downarrow \sigma(x)} \text{ if } \sigma(x) \text{ is defined} \qquad \text{(Prod)}\ \frac{\langle a_1, \sigma\rangle \downarrow n_1 \quad \langle a_2, \sigma\rangle \downarrow n_2}{\langle a_1 * a_2, \sigma\rangle \downarrow n'} \text{ if } n' = n_1 \cdot n_2$$

$$\text{(Diff)}\ \frac{\langle a_1, \sigma\rangle \downarrow n_1 \quad \langle a_2, \sigma\rangle \downarrow n_2}{\langle a_1 - a_2, \sigma\rangle \downarrow n'} \text{ if } n' = n_1 - n_2$$

The axiom (AxNum) states that a number evaluates to a number, with axiom (AxLoc) the value of a location is looked up (if it is defined), the rules (Sum), (Prod), (Diff) compute the sum, product, or difference of two numbers. Note that for instance, $a_1 + a_2$ is syntax of the programming language, while $n' = n_1 + n_2$ is meta-notation.

6.2.1.2. Rules for Evaluation of Boolean Expressions

The rules for evaluation of boolean expressions are:

$$\text{(AxT)}\ \frac{}{\langle \texttt{True}, \sigma\rangle \downarrow \texttt{True}} \qquad \text{(Eq)}\ \frac{\langle a_1, \sigma\rangle \downarrow n \quad \langle a_2, \sigma\rangle \downarrow m}{\langle a_1 = a_2, \sigma\rangle \downarrow \texttt{True}} \text{ if } n = m$$

$$\text{(AxF)}\ \frac{}{\langle \texttt{False}, \sigma\rangle \downarrow \texttt{False}} \qquad \text{(NEq)}\ \frac{\langle a_1, \sigma\rangle \downarrow n \quad \langle a_2, \sigma\rangle \downarrow m}{\langle a_1 = a_2, \sigma\rangle \downarrow \texttt{False}} \text{ if } n \neq m$$

$$\text{(Leq)}\ \frac{\langle a_1, \sigma\rangle \downarrow n \quad \langle a_2, \sigma\rangle \downarrow m}{\langle a_1 \leq a_2, \sigma\rangle \downarrow \texttt{True}} \text{ if } n \leq m \qquad \text{(NLeq)}\ \frac{\langle a_1, \sigma\rangle \downarrow n \quad \langle a_2, \sigma\rangle \downarrow m}{\langle a_1 \leq a_2, \sigma\rangle \downarrow \texttt{False}} \text{ if } n > m$$

$$\text{(AndT)}\ \frac{\langle b_1, \sigma\rangle \downarrow \texttt{True} \quad \langle b_2, \sigma\rangle \downarrow \texttt{True}}{\langle b_1 \wedge b_2, \sigma\rangle \downarrow \texttt{True}} \qquad \text{(AndF1)}\ \frac{\langle b_1, \sigma\rangle \downarrow \texttt{False}}{\langle b_1 \wedge b_2, \sigma\rangle \downarrow \texttt{False}}$$

$$\text{(AndF2)}\ \frac{\langle b_1,\sigma\rangle \downarrow \texttt{False} \quad \langle b_2,\sigma\rangle \downarrow \texttt{False}}{\langle b_1 \wedge b_2,\sigma\rangle \downarrow \texttt{False}} \qquad \text{(OrF)}\ \frac{\langle b_1,\sigma\rangle \downarrow \texttt{False} \quad \langle b_2,\sigma\rangle \downarrow \texttt{False}}{\langle b_1 \vee b_2,\sigma\rangle \downarrow \texttt{False}}$$

$$\text{(OrT1)}\ \frac{\langle b_1,\sigma\rangle \downarrow \texttt{True}}{\langle b_1 \vee b_2,\sigma\rangle \downarrow \texttt{True}} \qquad \text{(OrT2)}\ \frac{\langle b_1,\sigma\rangle \downarrow \texttt{False} \quad \langle b_2,\sigma\rangle \downarrow \texttt{True}}{\langle b_1 \vee b_2,\sigma\rangle \downarrow \texttt{True}}$$

$$\text{(Not1)}\ \frac{\langle b,\sigma\rangle \downarrow \texttt{False}}{\langle \neg b,\sigma\rangle \downarrow \texttt{True}} \qquad \text{(Not2)}\ \frac{\langle b,\sigma\rangle \downarrow \texttt{True}}{\langle \neg b,\sigma\rangle \downarrow \texttt{False}}$$

Axioms (AxT) and (AxF) state that a boolean value evaluates to a boolean value. Rules (Eq) and (Neq) evaluate the boolean value of an equality test on arithmetic expressions. Note that in the premise the evaluation of arithmetic expressions is used. Rules (Leq) and (NLeq) evaluate an $\leq$-operator. Rules (Not1) and (Not2) evaluate the negation. Rules (AndT), (AndF1), and (AndF2) evaluate a conjunction, rules (OrT), (OrF1), (OrF2) evaluate a disjunction. Note that conjunction and disjunction are evaluated "sequentially", i.e. if the value of can be calculated when the left value is available, the right value is not computed. This means $\langle \texttt{True} \vee b,\sigma\rangle \downarrow \texttt{True}$ and $\langle \texttt{False} \wedge b,\sigma\rangle \downarrow \texttt{False}$ for every b, in particular when b is undefined.

Example 6.2.2. *For $\sigma = \{x \mapsto 10, y \mapsto 7, z \mapsto 8\}$, we can built the derivation tree for the boolean expression $x \leq y + 4 \vee \texttt{True}$ as follows*

$$\dfrac{\dfrac{\dfrac{}{\langle x,\sigma\rangle \downarrow 10}\,\text{AxNum} \quad \dfrac{\dfrac{}{\langle y,\sigma\rangle \downarrow 7}\,\text{AxLoc} \quad \dfrac{}{\langle 4,\sigma\rangle \downarrow 4}\,\text{AxNum}}{\langle y+4,\sigma\rangle \downarrow 11}\,\text{Sum}\ \text{if } 11 = 7+4}{\langle x \leq y+4,\sigma\rangle \downarrow \texttt{True}}\,\text{Leq}\ \text{if } 10 \leq 11}{\langle x \leq y+4 \vee \texttt{True},\sigma\rangle \downarrow \texttt{True}}\,\text{OrT1}$$

The construction is done bottom-up, until the top of the tree consists of axioms and thus no more premises have to be shown.

Note that the semantics does not prescribe an exact order of evaluation (for instance in $a_1 + a_2$, the semantics does not fix the order of evaluation a_1 and a_2). This is a typical characteristics of a big-step semantics – it leaves some freedom in the implementation.

We could change this, by either fixing the order as an instruction in the premise of the rules, or by changing the rules, such that they distinguish between expressions and values, for instance, replacing the rule (Sum) by the following two rules would enforce left to right evaluation of addition:

$$\frac{\langle a_1,\sigma\rangle \downarrow n \quad \langle n + a_2,\sigma\rangle \downarrow m}{\langle a_1 + a_2,\sigma\rangle \downarrow m} \qquad \frac{\langle a_2,\sigma\rangle \downarrow n}{\langle m + a_2,\sigma\rangle \downarrow n'}\ \text{if } n' = m + n$$

6.2.1.3. Rules for Evaluation of Commands

The rules for evaluation of commands have side-effects, i.e. they modify the state σ. We write $\sigma[m/x]$ for the state σ where the value of x is changed to m, i.e.

$$\sigma[m/x](y) = \begin{cases} \sigma(x) & \text{if } y \neq x \\ m & \text{if } y = x \end{cases}$$

The evaluation rules are:

$$\text{(AxSkip)}\ \frac{}{\langle \texttt{skip}, \sigma\rangle \downarrow \sigma} \qquad \text{(Asgn)}\ \frac{\langle a, \sigma\rangle \downarrow m}{\langle x := a, \sigma\rangle \downarrow \sigma[m/x]} \qquad \text{(Seq)}\ \frac{\langle c_1, \sigma\rangle \downarrow \sigma' \quad \langle c_2, \sigma'\rangle \downarrow \sigma''}{\langle c_1; c_2, \sigma\rangle \downarrow \sigma''}$$

$$\text{(IfT)}\ \frac{\langle b, \sigma\rangle \downarrow \texttt{True} \quad \langle c_1, \sigma\rangle \downarrow \sigma'}{\langle \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi}, \sigma\rangle \downarrow \sigma'} \qquad \text{(IfF)}\ \frac{\langle b, \sigma\rangle \downarrow \texttt{False} \quad \langle c_2, \sigma\rangle \downarrow \sigma'}{\langle \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi}, \sigma\rangle \downarrow \sigma'}$$

$$\text{(WhileF)}\ \frac{\langle b, \sigma\rangle \downarrow \texttt{False}}{\langle \texttt{while } b \texttt{ do } c \texttt{ od}, \sigma\rangle \downarrow \sigma} \qquad \text{(WhileT)}\ \frac{\langle b, \sigma\rangle \downarrow \texttt{True} \qquad \langle c, \sigma\rangle \downarrow \sigma' \quad \langle \texttt{while } b \texttt{ do } c \texttt{ od}, \sigma'\rangle \downarrow \sigma''}{\langle \texttt{while } b \texttt{ do } c \texttt{ od}, \sigma\rangle \downarrow \sigma''}$$

The axiom (AxSkip) states that evaluation of the `skip`-command does not change the state. The rule (Asgn) evaluates the assignment, by evaluating the right-hand side and then modifying the state. The rule evaluates sequential composition of programs. Note that the different states enforce left to right evaluation. Rules (IfT) and (IfF) evaluate an `if-then-else`-command where depending on the value of the boolean expression the `then`- or the `else`-branch is evaluated. Rule (WhileF) is used to finish the evaluation of a `while`-command if the condition evaluates to `False`. The rule (WhileT) treats the case that the `while`-condition is `True`: then the body of the loop is evaluated once, and the newly derived state is us to evaluate the `while`-command again.

Example 6.2.3. *We show some examples for the evaluation of commands.*

The evaluation of $c = x := 1; y := 2$ *in the state* $\{x \mapsto 2\}$ *is*

$$\text{(Seq)}\ \frac{\text{(Asgn)}\ \dfrac{\text{(AxNum)}\ \dfrac{}{\langle 1, \{x \mapsto 2\}\rangle \downarrow 1}}{\langle x := 1, \{x \mapsto 2\}\rangle \downarrow \{x \mapsto 1\}} \qquad \text{(Asgn)}\ \dfrac{\text{(AxNum)}\ \dfrac{}{\langle 2, \{x \mapsto 1\}\rangle \downarrow 2}}{\langle y := 2, \{x \mapsto 1\}\rangle \downarrow \{x \mapsto 1, y \mapsto 2\}}}{\langle x := 1; y := 2, \{x \mapsto 2\}\rangle \downarrow \{x \mapsto 1, y \mapsto 2\}}$$

The evaluation of $\texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ y := y + 1; x := x - 1\ \texttt{od}$ *in the state* $\{x \mapsto 2, y \mapsto 0\}$ *is*

$$\dfrac{\begin{array}{c}\langle \neg(x \leq 1), \{x \mapsto 2, y \mapsto 0\}\rangle \downarrow \texttt{True} \quad (a)\\ \langle y := y + 1; x := x - 1, \{x \mapsto 2, y \mapsto 0\}\rangle \downarrow \{x \mapsto 1, y \mapsto 1\} \quad (b)\\ \langle \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ y := y + 1; x := x - 1\ \texttt{od}, \{x \mapsto 1, y \mapsto 1\}\rangle \downarrow \{x \mapsto 1, y \mapsto 1\} \quad (c)\end{array}}{\langle \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ y := y + 1; x := x - 1\ \texttt{od}, \{x \mapsto 2, y \mapsto 0\}\rangle \downarrow \{x \mapsto 1, y \mapsto 1\}}\ (\text{WhileT})$$

where we show the derivations of three judgments (a), (b), (c) separately:

For (a):

$$\dfrac{\dfrac{\dfrac{}{\langle x, \{x \mapsto 2, y \mapsto 0\}\rangle \downarrow 2}\ (\text{AxLoc}) \quad \dfrac{}{\langle 1, \{x \mapsto 2, y \mapsto 0\}\rangle \downarrow 1}\ (\text{AxNum})}{\langle x \leq 1, \{x \mapsto 2, y \mapsto 0\}\rangle \downarrow \texttt{False}}\ (\text{NLeq})}{\langle \neg(x \leq 1), \sigma\rangle \downarrow \texttt{True}}\ (\text{Not1})$$

For (b):

$$\dfrac{\begin{array}{c}\langle y := y + 1, \{x \mapsto 2, y \mapsto 0\}\rangle \downarrow \{x \mapsto 2, y \mapsto 1\}\ (b1)\\ \langle x := x - 1, \{x \mapsto 2, y \mapsto 1\}\rangle \downarrow \{x \mapsto 1, y \mapsto 1\}\ (b2)\end{array}}{\langle y := y + 1; x := x - 1, \{x \mapsto 2, y \mapsto 0\}\rangle \downarrow \{x \mapsto 1, y \mapsto 1\}}\ (\text{Seq})$$

Again we show the premises (b1) and (b2) separately:

For (b1):

$$\dfrac{\dfrac{\dfrac{}{\langle y, \{x \mapsto 2, y \mapsto 0\}\rangle \downarrow 0}\ (\text{AxLoc}) \quad \dfrac{}{\langle 1, \{x \mapsto 2, y \mapsto 0\}\rangle \downarrow 1}\ (\text{AxNum})}{\langle y + 1, \{x \mapsto 2, y \mapsto 0\}\rangle \downarrow 1}\ (\text{Sum})}{\langle y := y + 1, \{x \mapsto 2, y \mapsto 0\}\rangle \downarrow \{x \mapsto 2, y \mapsto 1\}}\ (\text{Asgn})$$

For (b2):

$$\dfrac{\dfrac{\dfrac{}{\langle x, \{x \mapsto 2, y \mapsto 1\}\rangle \downarrow 2}\ (\text{AxLoc}) \quad \dfrac{}{\langle 1, \{x \mapsto 2, y \mapsto 1\}\rangle \downarrow 1}\ (\text{AxNum})}{\langle x - 1, \{x \mapsto 2, y \mapsto 1\}\rangle \downarrow 1}\ (\text{Diff})}{\langle x := x - 1, \{x \mapsto 2, y \mapsto 1\}\rangle \downarrow \{x \mapsto 1, y \mapsto 1\}}\ (\text{Asgn})$$

For (c)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x, \{x \mapsto 1, y \mapsto 1\}\rangle \downarrow 1}\,(\text{AxLoc}) \quad \dfrac{}{\langle 1, \{x \mapsto 1, y \mapsto 1\}\rangle \downarrow 1}\,(\text{AxNum})}{\langle (x \leq 1), \{x \mapsto 1, y \mapsto 1\}\rangle \downarrow \texttt{True}}\,(\text{Leq})}{\langle \neg(x \leq 1), \{x \mapsto 1, y \mapsto 1\}\rangle \downarrow \texttt{False}}\,(\text{Not2})}{\langle \texttt{while } \neg(x \leq 1) \texttt{ do } y := y + 1; x := x - 1 \texttt{ od}, \{x \mapsto 1, y \mapsto 1\}\rangle \downarrow \{x \mapsto 1, y \mapsto 1\}}\,(\text{WhileF})$$

Example 6.2.4. *Let σ be an arbitrary state. Trying to derive $\langle \texttt{while True do skip od}\rangle \downarrow \sigma'$ for some state σ' in a bottom-up manner starts with*

$$\dfrac{\dfrac{}{\langle \texttt{True}, \sigma\rangle \downarrow \texttt{True}}\,(\text{AxT}) \quad \dfrac{}{\langle \texttt{skip}, \sigma\rangle \downarrow \sigma}\,(\text{AxSkip}) \quad \dfrac{\vdots}{\langle \texttt{while True do skip od}, \sigma\rangle \downarrow \sigma'}}{\langle \texttt{while True do skip od}, \sigma\rangle \downarrow \sigma'}\,(\text{WhileT})$$

Now the third premise has to be shown (i.e. the derivation tree replacing the dots is missing). But the premise to be shown, is exactly the conclusion of the whole tree. Hence, the construction of the derivation tree will not finish and thus no derivation tree exists.

6.2.1.4. Evaluation is Deterministic

In this section we show that evaluation is deterministic, i.e. if $\langle c, \sigma\rangle \downarrow \sigma'$ and $\langle c, \sigma\rangle \downarrow \sigma''$, then $\sigma' = \sigma''$.

We first proof that evaluation of arithmetic expressions is deterministic

Lemma 6.2.5. *Let a be an arithmetic expression and $\sigma \in \Sigma$. If $\langle a, \sigma\rangle \downarrow n$ and $\langle a, \sigma\rangle \downarrow n'$ then $n = n'$.*

Proof. By structural induction on a. If a is a number m, then only (AxNum) is applicable which always results in $\langle m, \sigma\rangle \downarrow m$ and thus $n = n' = m$. If a is a storage location x, to derive number n and n' as in the claim of the lemma, only axiom (AxLoc) can be used. This shows that $\sigma(x)$ must be defined and that applying the axiom always results in $\langle x, \sigma\rangle \downarrow \sigma(x)$ and thus $n = n' = \sigma(x)$.

If $a = a_1 + a_2$, then the induction hypothesis shows that the derivations $\langle a_i, \sigma\rangle \downarrow n_i$ are deterministic. Then the rule (Sum) must result in $\langle a_1 + a_2, \sigma\rangle \downarrow m$ where $m = n_1 + n_2$. Thus $n = n' = m$ must hold. The cases $a = a_1 * a_2$ and $a = a_1 - a_2$ are completely analogous. □

Lemma 6.2.6. *Let b be a boolean expression, $\sigma \in \Sigma$ and $\langle b, \sigma\rangle \downarrow v_1$ and $\langle b, \sigma\rangle \downarrow v_2$ then $v_1 = v_2$*

Proof. By structural induction on b. If b is a boolean value, then only (AxT) or (AxF) is applicable which always results in $\langle b, \sigma\rangle \downarrow b$ and thus $v_1 = v_2 = b$. If b is a conjunction $b_1 \wedge b_2$,

then by the induction hypothesis the evaluations $\langle b_i, \sigma\rangle \downarrow w_i$ are deterministic. Since the rule (AndT) can only be applied if w_1, w_2 are both `True`, rule (AndF1) can only be applied if w_1 is `False`, and rule (AndF2) can only be applied if w_1 is `True` and w_2 is `False`, the rule application is unique and only depends on the values w_1, w_2 which shows $v_1 = v_2$.

If b is a disjunction $b_1 \vee b_2$, then by the induction hypothesis the evaluations $\langle b_i, \sigma\rangle \downarrow w_i$ are deterministic. Since the rule (OrF) can only be applied if w_1, w_2 are both `False`, rule (OrT1) can only be applied if w_1 is `True`, and rule (OrT2) can only be applied if w_1 is `False` and w_2 is `True`, the rule application is unique and only depends on the values w_1, w_2 which shows $v_1 = v_2$.

If b is a negation $\neg b'$, then by the induction hypothesis the evaluation $\langle b', \sigma\rangle \downarrow v$ is deterministic. Then either rule (Not1) or rule (Not2) is applicable (but not both). This shows $v_1 = v_2$.

If b is an equation $a_1 = a_2$, then by Lemma 6.2.5 the evaluations $\langle a_i, \sigma\rangle \downarrow n_i$ are deterministic. Then either rule (Eq) or rule (NEq) is applicable (but not both). This shows $v_1 = v_2$.

If b is an inequation $a_1 \leq a_2$, then by Lemma 6.2.5 the evaluations $\langle a_i, \sigma\rangle \downarrow n_i$ are deterministic. Then either rule (Leq) or rule (NLeq) is applicable (but not both). This shows $v_1 = v_2$. □

Proposition 6.2.7. *Let c be a command, $\sigma \in \Sigma$ and $\langle c, \sigma\rangle \downarrow \sigma_1$ and $\langle c, \sigma\rangle \downarrow \sigma_2$ then $\sigma_1 = \sigma_2$*

Proof. By the derivation of $\langle c, \sigma\rangle \downarrow \sigma_1$. If the derivation consists of one rule application, then this rule must be an axiom, and there exists only the axiom (AxSkip). The $\sigma = \sigma_1$ and $c =$ `skip`. Then also $\sigma_2 = \sigma$, since no other derivation rule is applicable for this case.

For the induction step, we consider the last rule that is applied in the derivation of $\langle c, \sigma\rangle \downarrow \sigma_1$, i.e. the rule at the bottom of the derivation tree.

If the rule is (Seq), then $c = c_1; c_2$, and the derivation step must be of the form $\langle c_1, \sigma\rangle \downarrow \sigma'$ and $\langle c_2, \sigma'\rangle \downarrow \sigma_1$ for some σ'. For the derivation $\langle c, \sigma\rangle \downarrow \sigma_2$, the final rule also must be the rule (Seq), since no other rule matches this case, i.e. there is some σ'' with $\langle c_1, \sigma\rangle \downarrow \sigma''$ and $\langle c_2, \sigma''\rangle \downarrow \sigma_2$. The induction hypothesis applied to the subderivations show that $\sigma' = \sigma''$ and thus $\sigma_1 = \sigma_2$. Thus the claim holds.

If the rule is (IfT), then $c =$ `if` b `then` c_1 `else` c_2 `fi` and $\langle b, \sigma\rangle \downarrow$ `True` and $\langle c_1, \sigma\rangle \downarrow \sigma_1$. Lemma 6.2.6 shows that $\langle b, \sigma\rangle \downarrow$ `False` is impossible. and thus also in the derivation of $\langle c, \sigma\rangle \downarrow \sigma_2$ rule (IfT) must be used, where $\langle c_1, \sigma\rangle \downarrow \sigma_2$. The induction hypothesis applied to $\langle c_1, \sigma\rangle \downarrow \sigma_1$ now shows $\sigma_1 = \sigma_2$.

The cases (IfF) and (WhileF) are similar, so we ommit them.

If (WhileT) is used, then $c =$ `while` b `do` c' `od` and there exists σ' with $\langle b, \sigma\rangle \downarrow$ `True`, $\langle c, \sigma\rangle \downarrow \sigma'$ and $\langle$`while` b `do` c' `od`$, \sigma'\rangle \downarrow \sigma_1$.

By Lemma 6.2.6, $\langle b, \sigma\rangle \downarrow$ `False` is impossible and thus for $\langle c, \sigma\rangle \downarrow \sigma_2$ also the rule (WhileT) must be the last used rule. Hence, there exists σ'' with $\langle c, \sigma\rangle \downarrow \sigma''$ and $\langle$`while` b `do` c' `od`$, \sigma''\rangle \downarrow \sigma_2$. The induction hypothesis shows that $\sigma' = \sigma''$ and thus also $\sigma_1 = \sigma_2$. □

Note that an induction on the structure of c is not possible, since in the rule (WhileT) the command of the conclusion occurs in the premise.

Since evaluation is deterministic, we can express the semantics of a command as a partial function that maps a state to a new state:

Definition 6.2.8. *Let c be a command, then $[\![c]\!]_{eval} : \Sigma \to \Sigma$ is the partial function such that*

$$[\![c]\!]_{eval}\sigma = \sigma' \text{ iff } \langle c, \sigma\rangle \downarrow \sigma'$$

Note that there are programs such that $[\![c]\!]_{eval}$ is undefined for all states. One such program is `while True do skip od`.

We can define the following equivalence $\sim$ on commands which expresses that programs are equivalent if there input-output behavior is the same:

Definition 6.2.9. *The relation $\sim$ is defined as $c_1 \sim c_2$ iff for all $\sigma \in \Sigma : [\![c_1]\!]_{eval}\sigma = [\![c_2]\!]_{eval}\sigma$*

Lemma 6.2.10. *The equivalence*

$$(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}) \sim (\texttt{if}\ b\ \texttt{then}\ c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi})$$

holds.

Proof. Let $\sigma \in \Sigma$. We consider three cases:

- $\langle b, \sigma\rangle \downarrow \texttt{False}$. Then $\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma\rangle \downarrow \sigma$, since:

$$\text{(WhileF)}\ \frac{\langle b, \sigma\rangle \downarrow \texttt{False}}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma\rangle \downarrow \sigma}$$

  Also $\langle \texttt{if}\ b\ \texttt{then}\ c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}, \sigma\rangle \downarrow \sigma$ holds, since:

$$\text{(IfF)}\ \frac{\langle b, \sigma\rangle \downarrow \texttt{False} \qquad \text{(AxSkip)}\ \dfrac{}{\langle \texttt{skip}, \sigma\rangle \downarrow \sigma}}{\langle \texttt{if}\ b\ \texttt{then}\ c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}, \sigma\rangle \downarrow \sigma}$$

  This shows $[\![\texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}]\!]_{eval}\sigma = [\![\texttt{if}\ b\ \texttt{then}\ c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}]\!]_{eval}\sigma = \sigma$ in this case.

- $\langle b, \sigma\rangle \downarrow \texttt{True}$. If $\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma\rangle \downarrow \sigma'$ then σ'' must exist such that $\langle c, \sigma\rangle \downarrow \sigma''$ and $\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma''\rangle \downarrow \sigma'$ must hold (by rule (WhileT)).

  Then the following derivation can be constructed:

$$\text{(IfT)}\ \frac{\langle b, \sigma\rangle \downarrow \texttt{True} \qquad \text{(Seq)}\ \dfrac{\langle c, \sigma\rangle \downarrow \sigma'' \quad \langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma''\rangle \downarrow \sigma'}{\langle c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma\rangle \downarrow \sigma'}}{\langle \texttt{if}\ b\ \texttt{then}\ c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}, \sigma\rangle \downarrow \sigma'}$$

If there does not exists any σ' such that $\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma\rangle \downarrow \sigma'$, then there doese not exist a σ'' with $\langle c, \sigma\rangle \downarrow \sigma''$ or $\langle c, \sigma\rangle \downarrow \sigma''$ and there does not exist a σ' such that $\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma''\rangle \downarrow \sigma'$. In both cases, there does not exist σ' such that $\langle \texttt{if}\ b\ \texttt{then}\ c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}, \sigma\rangle \downarrow \sigma'$, since both parts are required for the derivation tree.

If $\langle \texttt{if}\ b\ \texttt{then}\ c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}, \sigma\rangle \downarrow \sigma'$, then σ'' must exist, such that $\langle c, \sigma\rangle \downarrow \sigma''$, and $\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma''\rangle \downarrow \sigma'$. Then the following derivation can be constructed:

$$\text{(WhileT)}\ \frac{\langle b, \sigma\rangle \downarrow \texttt{True} \quad \langle c, \sigma\rangle \downarrow \sigma'' \quad \langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma''\rangle \downarrow \sigma'}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma''\rangle \downarrow \sigma'}$$

If there does not exist σ' with $\langle \texttt{if}\ b\ \texttt{then}\ c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}, \sigma\rangle \downarrow \sigma'$, then the reasoning is analogous to the previous case showing that there does not exist σ' with $\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma\rangle \downarrow \sigma'$.

This shows $[\![\texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}]\!]_{eval}\sigma = [\![\texttt{if}\ b\ \texttt{then}\ c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}]\!]_{eval}\sigma$ for all subcases of the case $\langle b, \sigma\rangle \downarrow \texttt{True}$.

- if $\langle b, \sigma\rangle \downarrow v$ does not hold for any v. Then there must be a storage location x that occurs in b, such that $\sigma(x)$ is not defined. Then $[\![\texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}]\!]_{eval}\sigma$ is not defined and also $[\![\texttt{if}\ b\ \texttt{then}\ c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}]\!]_{eval}\sigma$ is not defined. Thus the claim holds. □

Assume that $\sim_{init}$ is defined analogous to our definition of $\sim$ with the difference, that all variables are initialized with value 0 (i.e. $\sigma(x) = 0$, if σ does not define a value for x).

Then $\sim$ and $\sim_{init}$ are different equivalences: The equivalence $\texttt{if}\ b\ \texttt{then}\ \texttt{skip}\ \texttt{else}\ \texttt{skip}\ \texttt{fi} \sim_{init} \texttt{skip}$ holds for every boolean expression b, since $\langle b, \sigma\rangle$ will always evaluate to `True` or `False`.

In our semantics, there exists b such that $\texttt{if}\ b\ \texttt{then}\ \texttt{skip}\ \texttt{else}\ \texttt{skip}\ \texttt{fi} \not\sim \texttt{skip}$: choose $x = x$ for b and a σ such that $\sigma(x)$ is undefined. Then there does not exist σ' with $[\![\texttt{if}\ b\ \texttt{then}\ \texttt{skip}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}]\!]_{eval}\sigma = \sigma'$, since $\langle b, \sigma\rangle \downarrow \sigma$ does not hold. But $[\![\texttt{skip}]\!]_{eval}\sigma = \sigma$ holds by (AxSkip).

6.2.2. A Small-Step-Semantics of IMP

We define a small-step semantics $\rightarrow$ in form of a reduction semantics for IMP. This approach is very similar to the reduction semantics that we have defined for the lambda calculus.

We will define reduction rules and reduction contexts to fix the position where the next reduction step has to be applied (alternatively, a labeling algorithm could be used).

The relation $\rightarrow$ will operate on configurations, i.e. tuples $\langle t, \sigma\rangle$ where t is an arithmetic expression, a boolean expression, or a command and $\sigma \in \Sigma$.

Definition 6.2.11. *The reduction rules are*

$(skip)$ $\langle \texttt{skip}; c, \sigma\rangle \to \langle c, \sigma\rangle$
$(asgn)$ $\langle x := m, \sigma\rangle \to \langle \texttt{skip}, \sigma[m/x]\rangle$ *if* $m \in \mathbb{Z}$
(ifT) $\langle \texttt{if True then } c_1 \texttt{ else } c_2 \texttt{ fi}, \sigma\rangle \to \langle c_1, \sigma\rangle$
(ifF) $\langle \texttt{if False then } c_1 \texttt{ else } c_2 \texttt{ fi}, \sigma\rangle \to \langle c_2, \sigma\rangle$
$(while)$ $\langle \texttt{while } b \texttt{ do } c \texttt{ od}, \sigma\rangle \to \langle \texttt{if } b \texttt{ then } c; \texttt{while } b \texttt{ do } c \texttt{ od else skip fi}, \sigma\rangle$
(sum) $\langle n + m, \sigma\rangle \to \langle n', \sigma\rangle$ *if* $n' = n + m$
$(prod)$ $\langle n * m, \sigma\rangle \to \langle n', \sigma\rangle$ *if* $n' = n \cdot m$
$(diff)$ $\langle n - m, \sigma\rangle \to \langle n', \sigma\rangle$ *if* $n' = n - m$
(loc) $\langle x, \sigma\rangle \to \langle n, \sigma\rangle$ *if* $\sigma(x) = n$
$(leqT)$ $\langle n \le m, \sigma\rangle \to \langle \texttt{True}, \sigma\rangle$ *if* $n \le m$
$(leqF)$ $\langle n \le m, \sigma\rangle \to \langle \texttt{False}, \sigma\rangle$ *if* $n > m$
(eqT) $\langle n = n, \sigma\rangle \to \langle \texttt{True}, \sigma\rangle$ *if* $n = m$
(eqF) $\langle n = m, \sigma\rangle \to \langle \texttt{False}, \sigma\rangle$ *if* $n \neq m$
(orT) $\langle \texttt{True} \vee b, \sigma\rangle \to \langle \texttt{True}, \sigma\rangle$
(orF) $\langle \texttt{False} \vee v, \sigma\rangle \to \langle v, \sigma\rangle$ *if* $v \in \{\texttt{True}, \texttt{False}\}$
$(andF)$ $\langle \texttt{False} \wedge b, \sigma\rangle \to \langle \texttt{False}, \sigma\rangle$
$(andT)$ $\langle \texttt{True} \wedge v, \sigma\rangle \to \langle v, \sigma\rangle$ *if* $v \in \{\texttt{True}, \texttt{False}\}$
$(notT)$ $\langle \neg\texttt{True}, \sigma\rangle \to \langle \texttt{False}, \sigma\rangle$
$(notF)$ $\langle \neg\texttt{False}, \sigma\rangle \to \langle \texttt{True}, \sigma\rangle$

We define three classes of reduction contexts

$$
\begin{aligned}
R_A &::= [\cdot] \mid R_A + a \mid R_A * a \mid R_A - a \mid n + R_A \mid n * R_A \mid n - R_A \\
R_B &::= [\cdot] \mid R_B \vee b \mid R_B \wedge b \mid \texttt{False} \vee R_B \mid \texttt{True} \wedge R_B \mid \neg R_B \\
&\quad \mid R_A \le a \mid n \le R_A \mid R_A = a \mid n = R_A \\
R_C &::= [\cdot] \mid R_C; c \mid \texttt{if } R_B \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi} \mid x := R_A
\end{aligned}
$$

Definition 6.2.12. *We define the reduction relation* $\xrightarrow{eval}$*:*

If $\langle s, \sigma\rangle \to \langle s', \sigma'\rangle$ *then for every* R_C*-context:* $\langle R_C[s], \sigma\rangle \xrightarrow{eval} \langle R_C[s'], \sigma'\rangle$.

To specify the used rule, we also write $\xrightarrow{eval, rule}$ *where* $rule$ *is the name of the used rule.*

Alternatively, we can use the following labeling algorithm to find the position for the next reduction: For a command c it starts with $c^\star$ and exhaustively applies the following shifting

rules:

$$
\begin{array}{lcl}
(c_1; c_2)^\star & \Rightarrow & (c_1^\star; c_2) \\
(X := a)^\star & \Rightarrow & (x := a^\star) \\
\texttt{if } b \texttt{ then } c \texttt{ else } c' \texttt{ fi}^\star & \Rightarrow & \texttt{if } b^\star \texttt{ then } c \texttt{ else } c' \texttt{ fi} \\
(a_1 \oplus a_2)^\star & \Rightarrow & (a_1^\star \oplus a_2) \text{ if } \oplus \in \{+, -, *, =, \leq\} \\
(n^\star \oplus a) & \Rightarrow & (n \oplus a^\star) \text{ if } n \in \mathbb{Z} \text{ and } \oplus \in \{+, -, *, =, \leq\} \\
(b_1 \vee b_2)^\star & \Rightarrow & (b_1^\star \vee b_2) \\
(b_1 \wedge b_2)^\star & \Rightarrow & (b_1^\star \wedge b_2) \\
(\neg\, b)^\star & \Rightarrow & (\neg\, b^\star) \\
(\texttt{False}^\star \vee b) & \Rightarrow & (\texttt{False} \vee b^\star) \\
(\texttt{True}^\star \wedge b) & \Rightarrow & (\texttt{True} \wedge b^\star)
\end{array}
$$

The reduction rules with labels are C is an arbitrary context that is a command and the hole may be at at command-, arithmetic expression-, or boolean expression-position.

$$
\begin{array}{l}
\langle C[\texttt{skip}^\star; c], \sigma\rangle \xrightarrow{eval, skip} \langle C[c], \sigma\rangle \\
\langle C[x := m^\star], \sigma\rangle \xrightarrow{eval, asgn} \langle C[\texttt{skip}], \sigma[m/x]\rangle \text{ if } m \in \mathbb{Z} \\
\langle C[\texttt{if True}^\star \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi}], \sigma\rangle \xrightarrow{eval, ifT} \langle C[c_1], \sigma\rangle \\
\langle C[\texttt{if False}^\star \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi}], \sigma\rangle \xrightarrow{eval, ifF} \langle C[c_2], \sigma\rangle \\
\langle C[\texttt{while } b \texttt{ do } c \texttt{ od}], \sigma\rangle^\star \xrightarrow{eval, while} \langle C[\texttt{if } b \texttt{ then } c; \texttt{while } b \texttt{ do } c \texttt{ od else skip fi}], \sigma\rangle \\
\langle C[n + m^\star], \sigma\rangle \xrightarrow{eval, sum} \langle C[n'], \sigma\rangle \text{ if } n' = n + m \\
\langle C[n * m^\star], \sigma\rangle \xrightarrow{eval, prod} \langle C[n'], \sigma\rangle \text{ if } n' = n \cdot m \\
\langle C[n - m^\star], \sigma\rangle \xrightarrow{eval, diff} \langle C[n'], \sigma\rangle \text{ if } n' = n - m \\
\langle C[x^\star], \sigma\rangle \xrightarrow{eval, loc} \langle C[n], \sigma\rangle \text{ if } \sigma(x) = n \\
\langle C[n \leq m^\star], \sigma\rangle \xrightarrow{eval, leqT} \langle C[\texttt{True}], \sigma\rangle \text{ if } n \leq m \\
\langle C[n \leq m^\star], \sigma\rangle \xrightarrow{eval, leqF} \langle C[\texttt{False}], \sigma\rangle \text{ if } n > m \\
\langle C[n = n^\star], \sigma\rangle \xrightarrow{eval, eqT} \langle C[\texttt{True}], \sigma\rangle \text{ if } n = m \\
\langle C[n = m^\star], \sigma\rangle \xrightarrow{eval, eqF} \langle C[\texttt{False}], \sigma\rangle \text{ if } n \neq m \\
\langle C[\texttt{True}^\star \vee b], \sigma\rangle \xrightarrow{eval, orT} \langle C[\texttt{True}], \sigma\rangle \\
\langle C[\texttt{False} \vee v^\star], \sigma\rangle \xrightarrow{eval, orF} \langle C[v], \sigma\rangle \text{ if } v \in \{\texttt{True}, \texttt{False}\} \\
\langle C[\texttt{False}^\star \wedge b], \sigma\rangle \xrightarrow{eval, andF} \langle C[\texttt{False}], \sigma\rangle \\
\langle C[\texttt{True} \wedge v^\star], \sigma\rangle \xrightarrow{eval, andT} \langle C[v], \sigma\rangle \text{ if } v \in \{\texttt{True}, \texttt{False}\} \\
\langle C[\neg\texttt{True}^\star], \sigma\rangle \xrightarrow{eval, notT} \langle C[\texttt{False}], \sigma\rangle \\
\langle C[\neg\texttt{False}^\star], \sigma\rangle \xrightarrow{eval, notF} \langle C[\texttt{True}], \sigma\rangle
\end{array}
$$

It is possible to verify that both definitions (i.e. using the labeling algorithm or using reduction

contexts) indeed lead to the same relation $\xrightarrow{eval}$. The proof requires the inspection of all syntactic cases. We omit the proof.

We write $\xrightarrow{eval,n}$ for n $\xrightarrow{eval}$-steps (i.e. the n-fold composition of $\xrightarrow{eval}$) and $\xrightarrow{eval,+}$ for the transitive closure and $\xrightarrow{eval,*}$ for the reflexive-transitive closure of $\xrightarrow{eval}$.

Small-step evaluation successfully stops if the configuration $\langle \texttt{skip}, \sigma \rangle$ for some $\sigma \in \Sigma$ is reached.

For command c and environment σ, we write $\langle c, \sigma \rangle \downarrow_{eval} \sigma'$ iff $\langle c, \sigma \rangle \xrightarrow{eval*} \langle \texttt{skip}, \sigma' \rangle$.

Note that there are configurations that have no successor w.r.t. $\xrightarrow{eval}$ but are not successful: this hold for configurations of the form $\langle R_C[x], \sigma \rangle$ where $\sigma(x)$ is undefined.

By inspecting all syntactic cases one can verify:

Lemma 6.2.13. *The reduction relation $\xrightarrow{eval}$ deterministic, i.e. if $\langle c, \sigma \rangle \xrightarrow{eval} \langle c', \sigma' \rangle$ and $\langle c, \sigma \rangle \xrightarrow{eval} \langle c'', \sigma'' \rangle$, then $c' = c''$ and $\sigma' = \sigma''$.*

Example 6.2.14. *We evaluate the command* $\texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}$ *for the state* $\{x \mapsto 3\}$*:*

$$
\begin{aligned}
& \langle \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}, \{x \mapsto 3\} \rangle \\
\xrightarrow{eval,while} & \langle \texttt{if}\ \neg(x \leq 1)\ \texttt{then}\ x := x - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}\ \texttt{else skip fi}, \{x \mapsto 3\} \rangle \\
\xrightarrow{eval,loc} & \langle \texttt{if}\ \neg(3 \leq 1)\ \texttt{then}\ x := x - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}\ \texttt{else skip fi}, \{x \mapsto 3\} \rangle \\
\xrightarrow{eval,leqF} & \langle \texttt{if}\ \neg\texttt{False}\ \texttt{then}\ x := x - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}\ \texttt{else skip fi}, \{x \mapsto 3\} \rangle \\
\xrightarrow{eval,notF} & \langle \texttt{if True then}\ x := x - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}\ \texttt{else skip fi}, \{x \mapsto 3\} \rangle \\
\xrightarrow{eval,ifT} & \langle x := x - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}, \{x \mapsto 3\} \rangle \\
\xrightarrow{eval,loc} & \langle x := 3 - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}, \{x \mapsto 3\} \rangle \\
\xrightarrow{eval,diff} & \langle x := 2; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}, \{x \mapsto 3\} \rangle \\
\xrightarrow{eval,asgn} & \langle skip; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}, \{x \mapsto 2\} \rangle \\
\xrightarrow{eval,skip} & \langle \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}, \{x \mapsto 2\} \rangle \\
\xrightarrow{eval,while} & \langle \texttt{if}\ \neg(x \leq 1)\ \texttt{then}\ x := x - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}\ \texttt{else skip fi}, \{x \mapsto 2\} \rangle \\
\xrightarrow{eval,loc} & \langle \texttt{if}\ \neg(2 \leq 1)\ \texttt{then}\ x := x - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}\ \texttt{else skip fi}, \{x \mapsto 2\} \rangle \\
\xrightarrow{eval,leqF} & \langle \texttt{if}\ \neg\texttt{False}\ \texttt{then}\ x := x - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}\ \texttt{else skip fi}, \{x \mapsto 2\} \rangle \\
\xrightarrow{eval,notF} & \langle \texttt{if True then}\ x := x - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}\ \texttt{else skip fi}, \{x \mapsto 2\} \rangle \\
\xrightarrow{eval,ifT} & \langle x := x - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}, \{x \mapsto 2\} \rangle \\
\xrightarrow{eval,loc} & \langle x := 2 - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}, \{x \mapsto 2\} \rangle \\
\xrightarrow{eval,diff} & \langle x := 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}, \{x \mapsto 2\} \rangle \\
\xrightarrow{eval,asgn} & \langle skip; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}, \{x \mapsto 1\} \rangle \\
\xrightarrow{eval,skip} & \langle \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}], \{x \mapsto 1\} \rangle \\
\xrightarrow{eval,while} & \langle \texttt{if}\ \neg(x \leq 1)\ \texttt{then}\ x := x - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}\ \texttt{else skip fi}, \{x \mapsto 1\} \rangle \\
\xrightarrow{eval,loc} & \langle \texttt{if}\ \neg(1 \leq 1)\ \texttt{then}\ x := x - 1; \texttt{while}\ \neg(x \leq 1)\ \texttt{do}\ x := x - 1\ \texttt{od}\ \texttt{else skip fi}, \{x \mapsto 1\} \rangle
\end{aligned}
$$

$$\xrightarrow{eval,leqT} \langle \texttt{if } \neg\texttt{True } \texttt{then } x := x-1; \texttt{while } \neg(x \leq 1) \texttt{ do } x := x-1 \texttt{ od else skip fi}, \{x \mapsto 1\}\rangle$$
$$\xrightarrow{eval,notT} \langle \texttt{if False then } x := x-1; \texttt{while } \neg(x \leq 1) \texttt{ do } x := x-1 \texttt{ od else skip fi}, \{x \mapsto 1\}\rangle$$
$$\xrightarrow{eval,ifF} \langle \texttt{skip}, \{x \mapsto 1\}\rangle$$

One can prove that the big-step semantics and the reduction semantics are equivalent. We will only sketch the proof. Two helpful lemmas are:

Lemma 6.2.15. *Let a be an arithmetic expression and σ be a state. Then $\langle a, \sigma\rangle \downarrow m$ iff $\langle a, \sigma\rangle \xrightarrow{n} \langle m, \sigma\rangle$ by applying the reduction rules of Definition 6.2.11.*

Proof. The "if"-direction can be shown by induction on the derivation tree for $\langle a, \sigma\rangle \downarrow m$, the "only-if"-direction can be shown by induction on the number n of steps. □

Lemma 6.2.16. *Let b be a boolean expression and σ be a state, $v \in \{\texttt{True}, \texttt{False}\}$. Then $\langle b, \sigma\rangle \downarrow v$ iff $\langle b, \sigma\rangle \xrightarrow{n} \langle v, \sigma\rangle$ by applying the reduction rules of Definition 6.2.11.*

Proof. The "if"-direction can be shown by induction on the derivation tree for $\langle b, \sigma\rangle \downarrow m$, the "only-if"-direction can be shown by induction on the number n of steps. Also Lemma 6.2.15 has to be used if arithmetic expressions are evaluated. □

Proposition 6.2.17. *For* IMP*-commands c and states $\sigma \in \Sigma$:*

$$\langle c, \sigma\rangle \downarrow_{eval} \sigma' \text{ iff } \langle c, \sigma\rangle \downarrow \sigma'$$

Proof. We consider the "if"-direction. If $\langle c, \sigma\rangle \xrightarrow{eval,n} \langle \texttt{skip}, \sigma'\rangle$ then $\langle c, \sigma\rangle \downarrow \sigma'$. We use induction on the number n of steps.

The base case is $n = 0$ and thus $c = \texttt{skip}$ and $\sigma' = \sigma$. Then (AxSkip) shows the claim.

If $n > 0$ i.e. $\langle c, \sigma\rangle \xrightarrow{eval} \langle c_1, \sigma_1\rangle \xrightarrow{eval,n-1} \langle \texttt{skip}, \sigma'\rangle$. The induction hypothesis shows that $\langle c_1, \sigma_1\rangle \downarrow \sigma'$.

Now all cases of the first reduction step (and c, c_1, σ, σ_1) have to be considered:

- If the step is a $\xrightarrow{eval,skip}$-step, then $c = \texttt{skip}; c_1$ and $\sigma = \sigma_1$. Then

$$\text{(Seq)}\ \frac{\text{(AxSkip)}\ \dfrac{}{\langle \texttt{skip}\rangle \downarrow \sigma} \quad \langle c_1, \sigma\rangle \downarrow \sigma'}{\langle \texttt{skip}; c_1, \sigma\rangle \downarrow \sigma'}$$

- If the step is a $\xrightarrow{eval,asgn}$-step, then there are two possible cases

1. $c = x := m$, $c_1 = \texttt{skip}$ and $\sigma_1 = \sigma[m/x] = \sigma'$. Then

$$\text{(Asgn)}\ \frac{\text{(AxNum)}\ \dfrac{}{\langle m, \sigma\rangle \downarrow m}}{\langle x := m, \sigma\rangle \downarrow \sigma[m/x]}$$

2. $c = x := m; c'$, $c_1 = \texttt{skip}; c'$, and $\sigma_1 = \sigma[m/x]$. Then $\langle \texttt{skip}; c', \sigma_1\rangle \xrightarrow{eval, skip} \langle c', \sigma_1\rangle$, and the induction hypothesis can also be applied to $\langle c', \sigma_1\rangle \xrightarrow{eval, n-2} \langle \texttt{skip}, \sigma'\rangle$ showing $\langle c', \sigma_1\rangle \downarrow \sigma'$. Then

$$\text{(Seq)}\ \frac{\text{(Asgn)}\ \dfrac{\text{(AxNum)}\ \dfrac{}{\langle m, \sigma\rangle \downarrow m}}{\langle x := m, \sigma\rangle \downarrow \sigma_1} \qquad \langle c', \sigma_1\rangle \downarrow \sigma'}{\langle x := m; c_1, \sigma\rangle \downarrow \sigma'}$$

- The cases that the reduction is a $\xrightarrow{eval, ifT}$-step or $\xrightarrow{eval, ifF}$-step, is similar to the previous one, we omit it.
- The case that the reduction is a $\xrightarrow{eval, while}$-step also has two subcases, depending on whether the while-command is a single command, or the first command of a sequence. We only consider the first case, the other case is similar (but requires the (Seq)-rule in the derivation tree).

  Let $c = \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}$, $c_1 = \texttt{if}\ b\ \texttt{then}\ c'; \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}$, and $\sigma_1 = \sigma$. The reduction semantics will evaluate b until it is a boolean value (otherwise a final configuration cannot be reached). We consider two cases:

  – b evaluates to $\texttt{False}$. Then $\langle \texttt{if}\ b\ \texttt{then}\ c'; \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}, \sigma\rangle \xrightarrow{eval, *} \langle \texttt{if}\ \texttt{False}\ \texttt{then}\ c'; \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}, \sigma\rangle \xrightarrow{eval, ifF} (\texttt{skip}, \sigma_2)$. Then also $\langle b, \sigma\rangle \xrightarrow{*} \langle \texttt{False}, \sigma\rangle$ (by omitting the outer reduction context) and by Lemma 6.2.16 $\langle b, \sigma\rangle \downarrow \langle \texttt{False}, \sigma\rangle$. This shows

$$\text{(WhileF)}\ \frac{\langle b, \sigma\rangle \downarrow \texttt{False}}{\langle \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}, \sigma\rangle \downarrow \sigma}$$

  – b evaluates to $\texttt{True}$. Then $\langle \texttt{if}\ b\ \texttt{then}\ c'; \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}, \sigma\rangle \xrightarrow{eval, *} \langle \texttt{if}\ \texttt{True}\ \texttt{then}\ c'; \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}, \sigma\rangle \xrightarrow{eval, ifT} (c'; \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}, \sigma)$. Then also $\langle b, \sigma\rangle \xrightarrow{*} \langle \texttt{True}, \sigma\rangle$ (by omitting the outer reduction context) and by Lemma 6.2.16 $\langle b, \sigma\rangle \downarrow \langle \texttt{False}, \sigma\rangle$. By the induction hypothesis we also have $\langle c'; \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}, \sigma\rangle \downarrow \sigma'$. Hence there must exist a derivation tree:

$$\text{(Seq)}\ \frac{\langle c', \sigma\rangle \downarrow \sigma_2 \langle \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}, \sigma_2\rangle \downarrow \sigma'}{\langle c'; \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}, \sigma\rangle \downarrow \sigma'}$$

and thus $\langle c', \sigma\rangle \downarrow \sigma_2$ and $\langle \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}, \sigma_2\rangle \downarrow \sigma'$ must hold.

Putting everything together this shows:

$$\text{(WhileT)}\ \frac{\langle b, \sigma\rangle \downarrow \texttt{True} \quad \langle c', \sigma\rangle \downarrow \sigma_2 \quad \langle \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}, \sigma_2\rangle \downarrow \sigma'}{\langle \texttt{while}\ b\ \texttt{do}\ c'\ \texttt{od}, \sigma\rangle \downarrow \sigma'}$$

- In all other cases the first reduction step operates on a boolean expression or an arithmetic expression (with an outer reduction context), We consider two cases:

  1. $c = R_c[\texttt{if}\ b\ \texttt{then}\ c'\ \texttt{else}\ c''\ \texttt{fi}]$ where b is a boolean expression. Then $\langle c, \sigma\rangle \xrightarrow{eval,k} \langle R_c[\texttt{if}\ v\ \texttt{then}\ c'\ \texttt{else}\ c''\ \texttt{fi}], \sigma\rangle \xrightarrow{eval,n-k} \langle \texttt{skip}, \sigma'\rangle$ with $v \in \{\texttt{True}, \texttt{False}\}$ and $k \geq 1$.

     Then also $\langle b, \sigma\rangle \xrightarrow{k} \langle v, \sigma\rangle$ with the reductions of Definition 6.2.11 and thus Lemmas 6.2.15 and 6.2.16 show $\langle b, \sigma\rangle \downarrow v$.

     Now we distinguish on v:

     - $v = \texttt{True}$: Then

       $$\langle R_c[\texttt{if}\ v\ \texttt{then}\ c'\ \texttt{else}\ c''\ \texttt{fi}], \sigma\rangle \xrightarrow{eval} \langle R_c[c'], \sigma\rangle \xrightarrow{eval,n-k-1} \langle \texttt{skip}, \sigma'\rangle$$

       Since $k > 0$ we can apply the induction hypothesis to $\langle R_c[c'], \sigma\rangle \xrightarrow{eval,n-k-1} \langle \texttt{skip}, \sigma'\rangle$ showing $\langle R_c[c'], \sigma\rangle \downarrow \sigma'$.

       If $R_c = [\cdot]$, then this shows

       $$\text{(IfT)}\ \frac{\langle b, \sigma\rangle \downarrow \texttt{True} \quad \langle c', \sigma\rangle \downarrow \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ c'\ \texttt{else}\ c''\ \texttt{fi}, \sigma\rangle \downarrow \sigma'}$$

       If $R_c = [\cdot]; c_0$ then $\langle R_c[c'], \sigma\rangle \downarrow \sigma'$ implies $\langle c', \sigma\rangle \downarrow \sigma_0$ and $\langle c_0, \sigma_0\rangle \downarrow \sigma'$ for some state σ_0.

       $$\text{(Seq)}\ \frac{\text{(IfT)}\ \dfrac{\langle b, \sigma\rangle \downarrow \texttt{True} \quad \langle c', \sigma\rangle \downarrow \sigma_0}{\langle \texttt{if}\ b\ \texttt{then}\ c'\ \texttt{else}\ c''\ \texttt{fi}, \sigma\rangle \downarrow \sigma_0} \quad \langle c_0, \sigma_0\rangle \downarrow \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ c'\ \texttt{else}\ c''\ \texttt{fi}; c_0, \sigma\rangle \downarrow \sigma'}$$

     - $v = \texttt{False}$: Then $\langle \texttt{if}\ v\ \texttt{then}\ c'\ \texttt{else}\ c''\ \texttt{fi}, \sigma\rangle \xrightarrow{eval} \langle c'', \sigma\rangle \xrightarrow{eval,n-k-1} \langle \texttt{skip}, \sigma'\rangle$. Since $k > 0$ we can apply the induction hypothesis to $\langle c'', \sigma\rangle \xrightarrow{eval,n-k-1} \langle \texttt{skip}, \sigma'\rangle$ showing $\langle c'', \sigma\rangle \downarrow \sigma'$.

Putting everything together shows

$$\text{(IfF)}\ \frac{\langle b, \sigma\rangle \downarrow \texttt{False} \quad \langle c', \sigma\rangle \downarrow \sigma'}{\langle R_c[\texttt{if } b \texttt{ then } c' \texttt{ else } c'' \texttt{ fi}], \sigma\rangle \downarrow \sigma'}$$

2. $c = R_c[x := a]$ where a is an arithmetic expression. This case can be proved analogously to the previous one.

For the "only if"-direction the induction has to be done on the derivation tree. We omit it. □

The language IMP is Turing complete. This can be shown by simulating a Turing machine with an IMP-program. In (Schöning, 2008) a similar language is shown to be Turing complete, by first showing that While-programs compute the same functions as Goto-programs and then shown Turing completeness of Goto-programs.

We only roughly sketch a direct proof of showing Turing completeness of IMP: for a Turing machine configuration wqw', the words w, w' and the state q are encoded as numbers to a base that is large enough to capture the tape alphabet and then encoded as integers. The three numbers are stored in storage locations $x_w, x_{w'}, x_q$ of the IMP-program. Operations of a Turing machine (i.e. replacing the current symbol and moving the read-/write-head) are operations on the numbers that can be implemented using division with reminders which can be expressed in IMP by using subtraction, addition, and multiplication. The state transition of Turing machine is encoded by a single while-loop: The while-condition checks whether x_q contains a state number that corresponds to an accepting state of the TM. If yes, the condition is false, the while-loop is finished and the IMP-program successfully stops. Otherwise the body of the while-loop performs one step of the TM. The body consists of nested `if` − `then` − `else`-commands: for each `if` b `then` c_1 `else` c_2 `fi`, c_2 contains the next `if` − `then` − `else` or `skip` if it is the last one. The condition b checks the state q stored in x_q and the first symbol a of w' stored in $x_{w'}$ and the code c_1 performs the transition defined by $\delta(q, a)$: it adjusts the content of x_w, x'_w, x_q accordingly.

This sketch shows that IMP-programs can simulated Turing machines and thus IMP is Turing complete. Note that the construction requires arbitrary large numbers. If the size of numbers where restricted, then IMP-programs would not be Turing complete.

6.2.3. An Abstract Machine as Operational Semantics of IMP

Defining the operational semantics in form of an abstract machine is a very explicit method to describe the semantics. However, it is machine-independent and the abstract machine can then be implemented on real machines. Often, an abstract machine can be implemented more efficiently than for instance the reduction semantics.

For reasoning, an abstract machine can also be used, but often the other formalisms are better suited.

We define an abstract machine for IMP.

Definition 6.2.18. *The* state *of the* IMP *machine is a triple* (E, T, S) *with*

- *Environment* E *that maps storage locations to numbers (similar to states* σ *used before).*
- *Current task* T *which might be a command or an (arithmetic or boolean) expression which is executed.*
- *Stack* S*: Contains numbers, boolean values (as intermediate results) and commands (which will be executed later). We use list notation for stacks and write* $s_1; s_2; \ldots; s_n$ *for a stack with* n *elements* $s_1, \ldots, s_n$ *where the top most element is* s_1*. We also write* $s_1; S$ *to denote a stack with top element* s_1 *and* S *is the remaining stack. With* $[]$ *we denote the empty stack.*

For a command c*, the* start state *of the* IMP *machine is the tuple* $(\emptyset, c, [])$*. For a given environment* E *we might also start the machine with* $(E, c, [])$*.*

A final state *of the* IMP *machine is any state of the form* $(E, \texttt{skip}, [])$*, i.e. the task is the command* `skip` *and the stack is empty. The result (or output) of the* IMP *machine is the environment* E *of the final state.*

The IMP machine may run infinitely if a non-terminating program is run. If the program uses undefined storage locations, the IMP machine will get stuck. This case is treated like non-termination.

Definition 6.2.19. *We define the transition relation* $\rightsquigarrow$ *of the* IMP *machine:*

$$
\begin{array}{lcl}
(E, (c_1; c_2), S) & \rightsquigarrow & (E, c_1, c_2; S) \\
(E, x := a, S) & \rightsquigarrow & (E, a, x :=; S) \\
(E, n, x :=; S) & \rightsquigarrow & (E[n/x], \texttt{skip}, S) \\
(E, x, S) & \rightsquigarrow & (E, n, S) \text{ if } E(x) = n \\
(E, \texttt{while } b \texttt{ do } c \texttt{ od}, S) & \rightsquigarrow & (E, b, [T : c; \texttt{while } b \texttt{ do } c \texttt{ od}, F : \texttt{skip}]; S) \\
(E, \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi}, S) & \rightsquigarrow & (E, b, [T : c_1, F : c_2]; S) \\
(E, \texttt{skip}, c; S) & \rightsquigarrow & (E, c, S) \\
(E, \texttt{True}, [T : c_1, F : c_2]; S) & \rightsquigarrow & (E, c_1, S) \\
(E, \texttt{False}, [T : c_1, F : c_2]; S) & \rightsquigarrow & (E, c_2, S)
\end{array}
$$

$$
\begin{array}{lcl}
(E, a_1 + a_2, S) & \rightsquigarrow & (E, a_1, (+a_2); S) \\
(E, n, (+a); S) & \rightsquigarrow & (E, a, (n+); S) \\
(E, m, (n+); S) & \rightsquigarrow & (E, m', S) \text{ if } m' = n + m \\
(E, a_1 - a_2, S) & \rightsquigarrow & (E, a_1, (-a_2); S) \\
(E, n, (-a); S) & \rightsquigarrow & (E, a, (n-); S) \\
(E, m, (n-); S) & \rightsquigarrow & (E, m', S) \text{ if } m' = n - m \\
(E, a_1 * a_2, S) & \rightsquigarrow & (E, a_1, (*a_2); S) \\
(E, n, (*a); S) & \rightsquigarrow & (E, a, (n*); S) \\
(E, m, (n*); S) & \rightsquigarrow & (E, m', S) \text{ if } m' = n \cdot m
\end{array}
$$

$$
\begin{array}{lcl}
(E, b_1 \wedge b_2, S) & \rightsquigarrow & (E, b_2, (\wedge\ b_2); S) \\
(E, \texttt{True}, (\wedge\ b_2); S) & \rightsquigarrow & (E, b_2, S) \\
(E, \texttt{False}, (\wedge\ b_2); S) & \rightsquigarrow & (E, \texttt{False}, S) \\
(E, b_1 \vee b_2, S) & \rightsquigarrow & (E, b_2, (\vee\ b_2); S) \\
(E, \texttt{True}, (\vee\ b_2); S) & \rightsquigarrow & (E, \texttt{True}, S) \\
(E, \texttt{False}, (\vee\ b_2); S) & \rightsquigarrow & (E, b_2, S) \\
(E, \neg b, S) & \rightsquigarrow & (E, b, \neg; S) \\
(E, \texttt{True}, \neg; S) & \rightsquigarrow & (E, \texttt{False}, S) \\
(E, \texttt{False}, \neg; S) & \rightsquigarrow & (E, \texttt{True}, S) \\
 & & \\
(E, a_1 = a_2, S) & \rightsquigarrow & (E, a_1, (=\ a_2); S) \\
(E, n, (=\ a); S) & \rightsquigarrow & (E, a, (n\ =); S) \\
(E, m, (n\ =); S) & \rightsquigarrow & (E, \texttt{True}, S) \text{ if } m = n \\
(E, m, (n\ =); S) & \rightsquigarrow & (E, \texttt{False}, S) \text{ if } m \neq n \\
(E, a_1 \leq a_2, S) & \rightsquigarrow & (E, a_1, (\leq\ a_2); S) \\
(E, n, (\leq\ a); S) & \rightsquigarrow & (E, a, (n \leq); S) \\
(E, m, (n \leq); S) & \rightsquigarrow & (E, \texttt{True}, S) \text{ if } n \leq m \\
(E, m, (n \leq); S) & \rightsquigarrow & (E, \texttt{False}, S) \text{ if } n > m
\end{array}
$$

The stack entries are:

- *commands c*
- *branches $[T : c_1, F : c_2]$ to continue the evaluation of a conditional or a while loop*
- *$x :=$ meaning that x has to be updated in the environment*
- *$(\oplus t)$ means that the current task evaluates the left argument of operator $\oplus$. If this task is finished the entry marks how to proceed. This is used for arithmetic and boolean expressions t and operators $\oplus \in \{+, -, *, =, \leq, \wedge, \vee\}$*
- *$\neg$ to negate the result of the current task*
- *$(n\oplus)$ means that the right argument of $\oplus$ is evaluated in the current task and after success the operator with n as first argument is applied. Here n is a number and $\oplus \in \{+, -, *, =, \leq\}$*

Example 6.2.20. *We consider the program $x := 2;$ `while` $2 \leq x$ `do` $x := x - 1$ `od` and evaluate it on the* IMP *machine:*

$$
\begin{array}{ll}
 & (\emptyset, x := 2; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}, [\,]) \\
\rightsquigarrow & (\emptyset, x := 2, \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}) \\
\rightsquigarrow & (\emptyset, 2, x :=; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}) \\
\rightsquigarrow & (\{x \mapsto 2\}, \texttt{skip}, \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}) \\
\rightsquigarrow & (\{x \mapsto 2\}, \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}, [\,]) \\
\rightsquigarrow & (\{x \mapsto 2\}, 2 \leq x, [T : x := x - 1; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}, F : \texttt{skip}]) \\
\rightsquigarrow & (\{x \mapsto 2\}, 2, \leq x; [T : x := x - 1; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}, F : \texttt{skip}]) \\
\rightsquigarrow & (\{x \mapsto 2\}, x, 2 \leq; [T : x := x - 1; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}, F : \texttt{skip}]) \\
\rightsquigarrow & (\{x \mapsto 2\}, 2, 2 \leq; [T : x := x - 1; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}, F : \texttt{skip}]) \\
\rightsquigarrow & (\{x \mapsto 2\}, \texttt{True}; [T : x := x - 1; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}, F : \texttt{skip}]) \\
\rightsquigarrow & (\{x \mapsto 2\}, x := x - 1; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}) \\
\rightsquigarrow & (\{x \mapsto 2\}, x - 1; x :=; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}) \\
\rightsquigarrow & (\{x \mapsto 2\}, x; -1; x :=; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}) \\
\rightsquigarrow & (\{x \mapsto 2\}, 2; -1; x :=; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}) \\
\rightsquigarrow & (\{x \mapsto 2\}, 1; 2-; x :=; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}) \\
\rightsquigarrow & (\{x \mapsto 2\}, 1; x :=; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}) \\
\rightsquigarrow & (\{x \mapsto 1\}, \texttt{skip}, \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}) \\
\rightsquigarrow & (\{x \mapsto 1\}, \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}, [\,]) \\
\rightsquigarrow & (\{x \mapsto 1\}, 2 \leq x, [T : x := x - 1; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}, F : \texttt{skip}]) \\
\rightsquigarrow & (\{x \mapsto 1\}, 2, \leq x[T : x := x - 1; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}, F : \texttt{skip}]) \\
\rightsquigarrow & (\{x \mapsto 1\}, x, 2 \leq [T : x := x - 1; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}, F : \texttt{skip}]) \\
\rightsquigarrow & (\{x \mapsto 1\}, 1, 2 \leq [T : x := x - 1; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}, F : \texttt{skip}]) \\
\rightsquigarrow & (\{x \mapsto 1\}, \texttt{False}; [T : x := x - 1; \texttt{while}\ 2 \leq x\ \texttt{do}\ x := x - 1\ \texttt{od}, F : \texttt{skip}]) \\
\rightsquigarrow & (\{x \mapsto 1\}, \texttt{skip}, [\,])
\end{array}
$$

Definition 6.2.21. *Let $\overset{*}{\rightsquigarrow}$ be the reflexive-transitive closure of $\rightsquigarrow$ and let $\overset{n}{\rightsquigarrow}$ be n steps of $\rightsquigarrow$. For a program c and an environment σ, we write $\langle c, \sigma\rangle \downarrow_{absm} \sigma'$ iff $(\sigma, c, \emptyset) \overset{*}{\rightsquigarrow} (\sigma', \texttt{skip}, [\,])$.*

Theorem 6.2.22. *The abstract machine is equivalent to the big-step semantics and to the reduction semantics, i.e. $\langle c, \sigma\rangle \downarrow_{absm} \sigma$ iff $\langle c, \sigma\rangle \downarrow_{eval} \sigma$ and $\langle c, \sigma\rangle \downarrow_{absm} \sigma$ iff $\langle c, \sigma\rangle \downarrow \sigma$ and*

Proof. It suffices to show the claim for the reduction semantics. It is straight-forward by an induction on the number of $\overset{eval}{\longrightarrow}$-steps for one direction, and another induction on the number of $\rightsquigarrow$-step for the other direction. We leave the full proof as an exercise. □

6.3. Denotational Semantics of IMP

In this section we define the denotational semantics of IMP. The idea is of the denotational semantics is to map each program construct to a mathematical object. Since the meaning of an

arithmetic or a boolean expression or a command, depends on the current state of the underlying programming, the mathematical objects itself are relations between states and number (boolean values, or states, resp.) Since evaluation in IMP is deterministic, these relations are in fact functions, and thus we use functions as objects, more specific, *partial* functions that map states to a number, a boolean value, or another state depending on the construct (arithmetic expressions, boolean expressions, commands). The functions must be partial, since for instance not every state defines the needed value of storage locations, or since non-termination of loops must also be treated as a function that is not defined for the start state.

We denote the mapping according to their syntax with $\mathcal{A}, \mathcal{B}$, and C for the denotation of arithmetic expressions, boolean expressions, and commands. We write the syntactic argument in $[\![\cdot]\!]$ brackets, and thus

- for arithmetic expressions a, $\mathcal{A}[\![a]\!] : \Sigma \to \mathbb{Z}$
- for boolean expression b, $\mathcal{B}[\![b]\!] : \Sigma \to \{\texttt{True}, \texttt{False}\}$
- for commands c, $C[\![c]\!] : \Sigma \to \Sigma$

where all images are partial functions.

Remark 6.3.1. *A partial function $f : M \to N$ is not necessarily defined for elements of M. The domain of a partial function f, is denoted as $Dom(f)$. It is the subset of M where f is defined (for a total function, $Dom(f) = M$ holds). The function f with $Dom(f) = \emptyset$ is never defined, and also called the empty function, written as $\emptyset$.*

We use λ-notation to define the denotation, where we write $\lambda\sigma \in \Sigma. \ldots$ to explicitly write that σ must be state. d

Definition 6.3.2 (Denotational Semantics of Arithmetic Expressions)**.**

$$
\begin{array}{lcl}
\mathcal{A}[\![n]\!] & := & \lambda\sigma \in \Sigma.n, \text{ if } n \in \mathbb{Z} \\
\mathcal{A}[\![x]\!] & := & \lambda\sigma \in \Sigma.\sigma(x) \text{ if } x \in Loc \\
\mathcal{A}[\![a_1 + a_2]\!] & := & \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_1]\!]\sigma) + (\mathcal{A}[\![a_2]\!]\sigma) \\
\mathcal{A}[\![a_1 - a_2]\!] & := & \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_1]\!]\sigma) - (\mathcal{A}[\![a_2]\!]\sigma) \\
\mathcal{A}[\![a_1 * a_2]\!] & := & \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_1]\!]\sigma) \cdot (\mathcal{A}[\![a_2]\!]\sigma)
\end{array}
$$

Note that if $\sigma(x)$ is not defined, then $\sigma \notin Dom(\lambda\sigma \in \Sigma.\sigma(x))$.

We assume for the denotation of boolean expressions, we also have boolean values `True`, `False` in the denotation with conjunction and disjunction that evaluates sequentially. We do not use different notation between the syntactic objects and the denotational objects.

Definition 6.3.3 (Denotational Semantics of Boolean Expressions)**.**

$$\begin{array}{lcl}
\mathcal{B}[\![\texttt{True}]\!] & := & \lambda\sigma \in \Sigma.\texttt{True} \\
\mathcal{B}[\![\texttt{False}]\!] & := & \lambda\sigma \in \Sigma.\texttt{False} \\
\mathcal{B}[\![a_1 = a_2]\!] & := & \lambda\sigma \in \Sigma.\mathcal{A}[\![a_1]\!]\sigma = \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{B}[\![a_1 \leq a_2]\!] & := & \lambda\sigma \in \Sigma.\mathcal{A}[\![a_1]\!]\sigma \leq \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{B}[\![\neg b]\!] & := & \lambda\sigma \in \Sigma.\neg(\mathcal{B}[\![b]\!]\sigma) \\
\mathcal{B}[\![b_1 \vee b_2]\!] & := & \lambda\sigma \in \Sigma.(\mathcal{B}[\![b_1]\!]\sigma) \vee (\mathcal{B}[\![b_2]\!]\sigma) \\
\mathcal{B}[\![b_1 \wedge b_2]\!] & := & \lambda\sigma \in \Sigma.(\mathcal{B}[\![b_1]\!]\sigma) \wedge (\mathcal{B}[\![b_2]\!]\sigma)
\end{array}$$

For the denotational semantics of commands, we introduce a helper function

$$\texttt{cond} : (\Sigma \to \mathit{Bool}) \to (\Sigma \to \Sigma) \to (\Sigma \to \Sigma) \to \Sigma \to \Sigma$$

defined as

$$(\texttt{cond}\ f\ g_1\ g_2)\ \sigma = \begin{cases} g_1\ \sigma, & \text{if } f\ \sigma = \texttt{True} \\ g_2\ \sigma, & \text{if } f\ \sigma = \texttt{False} \end{cases}$$

We also defined the identity:

$$\texttt{id}_\Sigma := \lambda\sigma \in \Sigma.\sigma$$

Now we can define the denotation of all commands except for while:

Definition 6.3.4 (Denotation of Commands (without `while`))**.**

$$\begin{array}{lcl}
C[\![\texttt{skip}]\!] & := & \mathit{id}_\Sigma \\
C[\![x := a]\!] & := & \lambda\sigma \in \Sigma.\sigma[(\mathcal{A}[\![a]\!]\sigma)/x] \\
C[\![c_0; c_1]\!] & := & C[\![c_1]\!] \circ C[\![c_0]\!] \\
 & = & \lambda\sigma \in \Sigma.(C[\![c_1]\!])(C[\![c_0]\!]\sigma) \\
C[\![\texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1\ \texttt{fi}]\!] & := & \lambda\sigma \in \Sigma.(\texttt{cond}\ (\mathcal{B}[\![b]\!])\ (C[\![c_0]\!])\ (C[\![c_1]\!]))\ \sigma
\end{array}$$

Note that $\sigma \notin \mathit{Dom}(C[\![x := a]\!])$ *if* $(\mathcal{A}[\![a]\!]\sigma)$ *is undefined.*

Defining the denotation of `while` is not as straight-forward as one could imagine. A first approach is to use the equivalence

$$\texttt{while}\ b\ \texttt{do}\ c_0\ \texttt{od} \ \sim\ \texttt{if}\ b\ \texttt{then}\ c_0; \texttt{while}\ b\ \texttt{do}\ c_0\ \texttt{od}\ \texttt{else}\ \texttt{skip}\ \texttt{fi}$$

This results in

$$C[\![\texttt{while}\ b\ \texttt{do}\ c_0\ \texttt{od}]\!] := \lambda\sigma \in \Sigma.(\texttt{cond}\ (\mathcal{B}[\![b]\!])\ (C[\![c_0; \texttt{while}\ b\ \texttt{do}\ c_0\ \texttt{od}]\!])\ \texttt{id}_\Sigma)\ \sigma$$

which can be simplified to

$$C[\![\texttt{while}\ b\ \texttt{do}\ c_0\ \texttt{od}]\!] := \texttt{cond}\ (\mathcal{B}[\![b]\!])\ (C[\![c_0; \texttt{while}\ b\ \texttt{do}\ c_0\ \texttt{od}]\!])\ \texttt{id}_\Sigma$$

Computing the denotation for the sequence c_0; `while` b `do` c_0 `od`, this results in

$$C[\![\texttt{while } b \texttt{ do } c_0 \texttt{ od}]\!] := \texttt{cond } (\mathcal{B}[\![b]\!])\ ((C[\![\texttt{while } b \texttt{ do } c_0 \texttt{ od}]\!]) \circ (C[\![c_0]\!]))\ \texttt{id}_\Sigma$$

Now we see a circularity: the left-hand side $C[\![\texttt{while } b \texttt{ do } c_0 \texttt{ od}]\!]$ of the defining equation occurs on the right-hand side. Thus, this is not a well-formed definition.

But we can use this circular description to get a solution. Let us assume, that we know the denotation of $C[\![\texttt{while } b \texttt{ do } c_0 \texttt{ od}]\!]$ and let us write φ for it (where $\varphi : \Sigma \to \Sigma$). Then the circularity tells us that φ must satisfy the property

$$\varphi = \texttt{cond } (\mathcal{B}[\![b]\!])\ (\varphi \circ (C[\![c_0]\!]))\ \texttt{id}_\Sigma$$

This equation can be rewritten as

$$\varphi = \Gamma(\varphi)$$

if we define

$$\Gamma := \lambda u \in (\Sigma \to \Sigma).\texttt{cond } (\mathcal{B}[\![b]\!])\ (u \circ (C[\![c_0]\!]))\ \texttt{id}_\Sigma$$

Note that $\Gamma : (\Sigma \to \Sigma) \to (\Sigma \to \Sigma)$, i.e. it takes a function of type $\Sigma \to \Sigma$ and returns a function of type $\Sigma \to \Sigma$.

In fact, we do not know the denotation φ, but now we know what it should be. The denotation of $C[\![\texttt{while } b \texttt{ do } c_0 \texttt{ od}]\!]$ is a *fixpoint* of Γ!

Indeed, we will use the least fixpoint of Γ, and to be correct we would need to prove that the fixpoint exists and that it can be constructed. We omit this part and refer to (Winskel, 1993) and (Stump, 2013), where the latter contains a lot of details and exact proofs.

Let us write $\texttt{Fix}(\Gamma)$ for the least fixpoint of Γ.

Definition 6.3.5 (Denotation of `while`)**.**

$$C[\![\texttt{while } b \texttt{ do } c_0 \texttt{ od}]\!] := \texttt{Fix}(\Gamma)$$

where $\Gamma := \lambda u : \Sigma \to \Sigma.\texttt{cond } (\mathcal{B}[\![b]\!])\ (u \circ (C[\![c_0]\!]))\ \texttt{id}_\Sigma$

Operationally, the idea of computing the least fixpoint is to compute the partial functions $F_n[\![\texttt{while } b \texttt{ do } c_0 \texttt{ od}]\!]$ which represent the denotation of the `while`-loop `while` b `do` c_0 `od` where only n iterations are allowed (for states σ that require more than n iterations F_n is undefined). The denotation $C[\![\texttt{while } b \texttt{ do } c_0 \texttt{ od}]\!]$ is then the union of all functions $F_n[\![\texttt{while } b \texttt{ do } c_0 \texttt{ od}]\!]$.

We can use the idea to make the computation of F_n explicit:

For $n = 0$, the function $F_0[\![\texttt{while } b \texttt{ do } c \texttt{ od}]\!]$ is only defined on those states σ where no iteration of the loop is necessary, i.e. these are the states σ where $\mathcal{B}[\![b]\!]\sigma = \texttt{False}$ holds. This shows

$$F_0[\![\texttt{while } b \texttt{ do } c \texttt{ od}]\!] = \{\sigma \mapsto \sigma \mid \mathcal{B}[\![b]\!](\sigma) = \texttt{False}\}$$

(Note that we use set notation for functions, where all mappings are written explicitly as elements $x \mapsto y$ whenever $f(x) = y$ should hold.),

For $n = 1$, the function $F_1[\![\texttt{while } b \texttt{ do } c \texttt{ od}]\!]$ is defined on those states σ where at most 1 iteration of the loop is necessary:

$$\begin{aligned} F_1[\![\texttt{while}\, b\, \texttt{do}\, c\, \texttt{od}]\!] = \quad & \{\sigma \mapsto \sigma \mid \mathcal{B}[\![b]\!](\sigma) = \texttt{False}\} \\ \cup \quad & \{\sigma \mapsto \sigma' \mid \mathcal{B}[\![b]\!](\sigma) = \texttt{True} \text{ and } C[\![c]\!](\sigma) = \sigma' \text{ and } \mathcal{B}[\![b]\!](\sigma') = \texttt{False}\} \end{aligned}$$

The general case for $n \geq 1$ can be written as:

$$\begin{aligned} F_n[\![\texttt{while}\, b\, \texttt{do}\, c\, \texttt{od}]\!] = \quad & F_{n-1}[\![\texttt{while } b \texttt{ do } c \texttt{ od}]\!] \\ \cup \quad & \{\sigma \mapsto \sigma' \mid F_{n-1}[\![\texttt{while } b \texttt{ do } c \texttt{ od}]\!](\sigma) = \sigma' \text{ and } \mathcal{B}[\![b]\!](\sigma') = \texttt{True} \\ & \quad \text{and } C[\![c]\!](\sigma') = \sigma'' \text{ and } \mathcal{B}[\![b]\!](\sigma'') = \texttt{False}\} \end{aligned}$$

Since $F_i[\![\texttt{while } b \texttt{ do } c \texttt{ od}]\!] \subseteq F_{i+1}[\![\texttt{while } b \texttt{ do } c \texttt{ od}]\!]$ for all $i \in \mathbb{N}_0$, the infinite union can be built, i.e. $C[\![\texttt{while } b \texttt{ do } c_0 \texttt{ od}]\!] = \bigcup_{n\in\mathbb{N}_0} F_n[\![\texttt{while } b \texttt{ do } c_0 \texttt{ od}]\!]$.

It can be shown that this union is a fixpoint of Γ and that it is the least fixpoint.

Another construction of the least fixpoint is the following: start with the smallest function and then iteratively apply Γ and union all results. This will compute the least fixpoint. The smallest function is the empty function $\emptyset$ which is undefined for all states. Let us write φ_i for the i-fold application of Γ to $\emptyset$, i.e. $\varphi_0 = \emptyset$ and $\varphi_i = \Gamma(\varphi_{i-1})$ for $i > 0$. Then

$$\texttt{Fix}(\Gamma) = \bigcup_{i\in\mathbb{N}_0} \varphi_i$$

This union can be built, since the chain $\varphi_0 \subseteq \varphi_1 \subseteq \varphi_2 \subseteq \ldots$ is increasing w.r.t. $\subseteq$.

Example 6.3.6. *We compute the denotation of* $\texttt{while } x = 0 \texttt{ do skip od}$*:*

$$C[\![\texttt{while } x = 0 \texttt{ do skip od}]\!] = \texttt{Fix}(\Gamma) \textit{ where}$$

$$\begin{aligned} \Gamma \quad &:= \lambda u \in \Sigma \to \Sigma.(\texttt{cond } (\mathcal{B}[\![x = 0]\!])\ (u \circ \texttt{id})\ \texttt{id}) \\ &= \lambda u \in \Sigma \to \Sigma.(\texttt{cond } (\lambda\sigma \in \Sigma.\sigma(x) = 0)\ u\ \texttt{id}) \end{aligned}$$

We compute $\varphi_0, \varphi_1, \ldots$.

- $\varphi_0 = \emptyset$.
- $\varphi_1 = \Gamma\varphi_0$. *Using the definition of* Γ*:*

  $$\varphi_1 = \Gamma\varphi_0 = \texttt{cond } (\lambda\sigma \in \Sigma.\sigma(x) = 0)\ \emptyset\ \texttt{id}$$

  This can be expressed as $\varphi_1 = \{\sigma \mapsto \sigma \mid x \in Dom(\sigma) \text{ and } \sigma(x) \neq 0\}$
- $\varphi_2 = \texttt{cond } (\lambda\sigma \in \Sigma.\sigma(x) = 0)\ \varphi_1\ \texttt{id}$

If $\sigma(x) \neq 0$, then it is `id` *and otherwise it is φ_1. This can be expressed as*

$$\varphi_2 = \{\sigma \mapsto \sigma \mid x \in Dom(\sigma) \text{ and } \sigma(x) \neq 0\} \cup \{\sigma \mapsto \varphi_1\sigma \mid \sigma(x) = 0\}$$

But $\{\sigma \mapsto \varphi_1\sigma \mid \sigma(x) = 0\} = \emptyset$, since $\varphi_1\sigma$ is undefined for $\sigma(x) = 0$. This shows $\varphi_2 = \varphi_1$.

- *Since $\varphi_2 = \varphi_1$, we also have $\varphi_i = \varphi_1$ for all $i \geq 1$.*

Thus $\texttt{Fix}(\Gamma) = \varphi_1 = \{\sigma \mapsto \sigma \mid x \in Dom(\sigma) \text{ and } \sigma(x) \neq 0\}$.

This matches the intuition that the program terminates (with unchanged state), if x is defined and $x \neq 0$ holds in the initial state.

6.4. Equivalence of Operational and Denotational Semantics

In this section, we prove that the operational semantics and the denotational semantics of IMP are equivalent. We will use the big-step semantics.

Lemma 6.4.1. *For all arithmetic expressions a and states $\sigma \in \Sigma$: $\mathcal{A}[\![a]\!]\sigma = n$ iff $\langle a, \sigma\rangle \downarrow n$*

Proof. We use structural induction on a.

- If $a = n$, then $\mathcal{A}[\![n]\!]\sigma = n$ and $\langle n, \sigma\rangle \downarrow n$ by axiom (AxNum).
- If $a = x$, then $\mathcal{A}[\![x]\!]\sigma = \sigma(x)$ and $\langle x, \sigma\rangle \downarrow \sigma(x)$ by axiom (AxLoc).
- If $a = a_1 + a_2$, then: $\mathcal{A}[\![a_1 + a_2]\!]\sigma = n = (\mathcal{A}[\![a_1]\!]\sigma) + (\mathcal{A}[\![a_2]\!]\sigma)$
  iff $(\mathcal{A}[\![a_1]\!]\sigma) = n_1$ and $(\mathcal{A}[\![a_2]\!]\sigma)$ and $n = n_1 + n_2$
  iff (by the induction hypothesis)
  $\langle a_1, \sigma\rangle \downarrow n_1$ and $\langle a_2, \sigma\rangle \downarrow n_2$
  iff (by the operational semantics, rule (Sum))
  $\langle a_1 + a_2, \sigma\rangle \downarrow n$
- The cases $a = a_1 - a_2$ and $a = a_1 * a_2$ are completely analogous to the previous one.

□

Lemma 6.4.2. *For all boolean expressions b, $\sigma \in \Sigma$, and $v \in \{\texttt{True}, \texttt{False}\}$:*

$$\mathcal{B}[\![b]\!]\sigma = v \text{ iff } \langle a, \sigma\rangle \downarrow v$$

Sketch. The proof is by structural induction on b. It is similar to the previous proof and uses Lemma 6.4.1 if b requires the value of an arithmetic expression. □

Theorem 6.4.3. *For all commands c of* IMP *and $\sigma \in \Sigma$: $C[\![c]\!]\sigma = \sigma'$ iff $\langle c, \sigma\rangle \downarrow \sigma'$*

Proof. We show the "if"-direction and omit the "only-if"-direction. The latter can be proved by induction on the derivation tree for $\langle c, \sigma\rangle \downarrow \sigma'$. For the "if"-direction, let us assume that $C[\![c]\!]\sigma = \sigma'$. We use structural induction on c:

- $C[\![\texttt{skip}]\!]\sigma = \sigma$ and $\langle \texttt{skip}, \sigma\rangle \downarrow \sigma$.
- $C[\![x := a]\!]\sigma = \sigma[\mathcal{A}[\![a]\!]\sigma/x]$. Assume that $\mathcal{A}[\![a]\!]\sigma = n$ Lemma 6.4.1 shows $\langle a, \sigma\rangle \downarrow n$. Thus we have $C[\![x := a]\!]\sigma = \sigma[n/x]$, and thus $\langle x := a, \sigma\rangle \downarrow \sigma[n/x]$.
- $C[\![c_0; c_1]\!]\sigma = C[\![c_1]\!](C[\![c_0]\!]\sigma) = \sigma'$. The induction hypothesis shows for all σ_1, σ_2, σ_3, and σ_4:
  - $C[\![c_0]\!]\sigma_1 = \sigma_2$ implies $\langle c_0, \sigma_1\rangle \downarrow \sigma_2$ and
  - $C[\![c_1]\!]\sigma_3 = \sigma_4$ implies $\langle c_0, \sigma_3\rangle \downarrow \sigma_4$.

  With $\sigma_1 = \sigma, \sigma_2 = \sigma_3 = C[\![c_0]\!]\sigma$ and $\sigma_4 = \sigma'$, rule (Seq) shows $\langle c_0; c_1, \sigma\rangle \downarrow \sigma''$.
- $C[\![\texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1 \texttt{ fi}]\!]\sigma = (\texttt{cond } \mathcal{B}[\![b]\!]\ C[\![c_0]\!]\ C[\![c_1]\!])\sigma = \sigma'$.
  If $(\mathcal{B}[\![b]\!]\sigma) = \texttt{True}$, then $\langle b, \sigma\rangle \downarrow \texttt{True}$ and if $(\mathcal{B}[\![b]\!]\sigma) = \texttt{False}$, then $\langle b, \sigma\rangle \downarrow \texttt{False}$ by Lemma 6.4.2. By the induction hypothesis $(C[\![c_0]\!]\sigma) = \sigma'$ implies $\langle c_0, \sigma\rangle \downarrow \sigma'$ and $(C[\![c_1]\!]\sigma) = \sigma'$ iff $\langle c_1, \sigma\rangle \to \sigma'$.

  Using rule (IfT) or (IfF), resp., this shows $\langle \texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1 \texttt{ fi}, \sigma\rangle \downarrow \sigma'$.
- $C[\![\texttt{while } b \texttt{ do } c_0 \texttt{ od}]\!] = \texttt{Fix}(\Gamma)$ where $\Gamma := \lambda u \in (\Sigma \to \Sigma).\texttt{cond } (\mathcal{B}[\![b]\!])\ (u \circ (C[\![c_0]\!]))\ \texttt{id}$. Then

$$\begin{aligned}\Gamma(\varphi) = \quad & \{\sigma \mapsto \sigma \mid (\mathcal{B}[\![b]\!]\sigma) = \texttt{False}\} \\ \cup \quad & \{\sigma \mapsto \sigma' \mid (\mathcal{B}[\![b]\!]\sigma) = \texttt{True} \text{ and } (\sigma \mapsto \sigma') \in \varphi \circ (C[\![c_0]\!])\}\end{aligned}$$

  Let $\varphi_n := \Gamma^n(\emptyset)$.
  Then:

$$\begin{aligned}\varphi_{n+1} = \quad & \{\sigma \mapsto \sigma \mid (\mathcal{B}[\![b]\!]\sigma) = \texttt{False}\} \\ \cup \quad & \{\sigma \mapsto \sigma' \mid (\mathcal{B}[\![b]\!]\sigma) = \texttt{True} \text{ and } (\sigma \mapsto \sigma') \in \varphi_n \circ (C[\![c_0]\!])\}\end{aligned}$$

  By induction on n, we show that $(\sigma \mapsto \sigma') \in \varphi_n \implies \langle \texttt{while } b \texttt{ do } c_0 \texttt{ od}, \sigma\rangle \downarrow \sigma'$.
  - $n = 0$: $\varphi_0 = \emptyset$, thus the left-hand side of the implication is false and thus the implication is true.
  - $n > 0$: Assume that the claim holds for n and let $(\sigma \mapsto \sigma') \in \varphi_{n+1}$. Then either $(\mathcal{B}[\![b]\!]\sigma) = \texttt{False}$ and $\sigma' = \sigma$. Then (by Lemma 6.4.2) $\langle b, \sigma\rangle \downarrow \texttt{False}$ and thus by rule (WhileF), $\langle \texttt{while } b \texttt{ do } c_0 \texttt{ od}, \sigma\rangle \downarrow \sigma$.

    If $\mathcal{B}[\![b]\!]\sigma = \texttt{True}$ then $\langle b, \sigma\rangle \downarrow \texttt{True}$ by Lemma 6.4.2. Since $(\sigma \mapsto \sigma') \in \varphi_{n+1}$, there exists σ'' with $(\sigma \mapsto \sigma'') \in C[\![c_0]\!]$ and $(\sigma'' \mapsto \sigma') \in \varphi_n$.

    By the outer induction hypothesis we get $\langle c_0, \sigma'\rangle \downarrow \sigma''$. By the inner induction hypothesis we get $\langle \texttt{while } b \texttt{ do } c_0 \texttt{ od}, \sigma''\rangle \downarrow \sigma'$. Rule (WhileT) shows $\langle \texttt{while } b \texttt{ do } (c_0; \texttt{while } b \texttt{ do } c_0 \texttt{ od}) \texttt{ od}, \sigma\rangle \downarrow \sigma'$.

  Since $\texttt{Fix}(\Gamma) = \bigcup \varphi_n$, we have:

$$(\sigma \mapsto \sigma') \in \texttt{Fix}(\Gamma) \implies (\sigma \mapsto \sigma') \in \varphi_n \text{ for some } n$$

  and thus $\langle \texttt{while } b \texttt{ do } c_0 \texttt{ od}, \sigma\rangle \downarrow \sigma'$.

□

Example 6.4.4. *Consider the loop*

$$(\texttt{while True do skip od})$$

The big-step operational semantics cannot derive any σ' such that $\langle \texttt{while True do skip od}, \sigma\rangle \downarrow \sigma'$ for any σ.

The small-step operational semantics has an infinite sequence of reduction steps:

$$\begin{aligned} & \langle \texttt{while True do skip od}, \sigma\rangle \\ \xrightarrow{eval, while} & \langle \texttt{if True then while True do skip od else skip fi}, \sigma\rangle \\ \xrightarrow{eval, ifT} & \langle \texttt{while True do skip od}, \sigma\rangle \\ \xrightarrow{eval, while} & \ldots \end{aligned}$$

The denotational semantics is

$C[\![\texttt{while True do skip od}]\!] := \texttt{Fix}(\Gamma)$ *where* $\Gamma := \lambda u \in (\Sigma \to \Sigma).\texttt{cond}\ (\lambda\sigma.\texttt{True})\ (u \circ \texttt{id})\ \texttt{id}$

For computing the fixpoint, we compute $\varphi_0, \varphi_1, \ldots$:

- $\varphi_0 = \emptyset$
- $\varphi_1 = \Gamma(\varphi_0) = \texttt{cond}\ (\lambda\sigma.\texttt{True})\ (\emptyset \circ \texttt{id})\ \texttt{id} = \texttt{cond}\ (\lambda\sigma.\texttt{True})\ \emptyset\ \texttt{id} = \emptyset$ *and thus* $\varphi_1 = \varphi_0$

Since $\varphi_1 = \varphi_0$, this shows $\varphi_i = \varphi_0 = \emptyset$ and thus $\texttt{Fix}(\Gamma) = \emptyset$. Thus the denotation is the partial function that is undefined for every state σ.

6.5. Conclusion and References

We have shown different concepts and formalisms for operational semantics and denotational semantics. As example we used a simple but Turing complete imperative language, called IMP. We proved equivalence of the denotational and the operational semantics.

As stated in the introduction, there are several books that explain different semantics for a small imperative language like IMP. Very similar languages are treated in (Winskel, 1993; Stump, 2013). The presentation of the content follows the lecture notes (Schmidt-Schauß, 2013).

7. Conclusion

We introduced several aspects on the foundations of programming languages, where we mainly considered core languages of real programming languages that are Turing-complete and follow different programming paradigms. The first chapters consider the core of functional languages, while in the final chapter our running example was an imperative language. Besides the operational semantics and denotational semantics we studied the typing of programs which sometimes is called its static semantics. Of course, these notes only cover some parts of the topic and can mainly be seen as an introduction or starting point for further studies.

A. Turing Machines in Haskell

```
-- ------------------------------------------------------------
-- Imports
import Data.List
import Data.Maybe
-- ------------------------------------------------------------

-- data types

-- The type Move represents the movement of the read/write-head

data Move = MLeft | MRight | MNothing
 deriving (Eq, Show)

-- The type TM represents a Turing machine.
-- The input alphabet, the tape alphabet, and the set of states
-- are only represented implicitely
-- The remaining components are:
--  * start: The start state of the Turing machine
--  * accepting: The set of accepting states of the Turing machine
--  * delta: The state transition function
-- The data type is polymorphic over the tape alphabet and the states.
-- The blank symbol is represented by using the Maybe-type:
-- - Nothing means a blank symbol
-- - Just s means the symbol s of the alphabet

data TM alphabet state =
 TM {
  start     :: state,
  accepting :: [state],
  -- delta receives the state and the current symbol and
  -- returns the new state, the new symbol,
  -- and the movement of read/write-head
  delta     :: (state,Maybe alphabet) -> (state,Maybe alphabet,Move)
  }

-- TMConfig is a data type for Turing machine configurations:
-- The current (explored) tape content is
--                  before ++ [current] ++ after,
-- the read/write-head is on the symbol current
```

```
-- currState is the current state of the machine

data TMConfig alphabet state =
 TMConfig {
  before    :: [Maybe alphabet],
  current   :: Maybe alphabet,
  after     :: [Maybe alphabet],
  currState :: state
 }
 deriving(Eq,Show)

-- ------------------------------------------------------------
-- Functions

-- oneStep calculates one step of the TM
-- oneStep receives a TM and a TMConfig
-- if the machine is already in accepting state,
-- oneStep returns Nothing,
-- otherwise, oneStep returns Just c, where
--   c is the configuration after performing one step

oneStep :: Eq s => TM a s -> (TMConfig a s) -> Maybe (TMConfig a s)
oneStep tm tc
 -- already in an accepting state:
 | (currState tc) `elem` (accepting tm) = Nothing
 -- not in an accepting state:
 | otherwise =
    case (delta tm) (currState tc, current tc) of
    (s',a',m) -> -- successor state, symbol, movement
      case m of
       MNothing -> Just $ tc {currState = s', current = a'}
       MRight ->
        if null (after tc) then
          Just $ tc {currState = s',
                     current = Nothing,
                     before = (before tc)++[a'],
                     after = []}
        else Just $ tc {currState = s',
                        current = head (after tc),
                        before = (before tc) ++ [a'],
                        after = tail (after tc)}
       MLeft ->
        if null (before tc) then
           error "move left on start position"
         else if (null (after tc)) && (isNothing a') then
                 Just $ tc {currState = s',
                            current = last (before tc),
```

```
                            before = init (before tc),
                            after = []}
               else
                  Just $ tc {currState = s',
                            current = last (before tc),
                            before = init (before tc),
                            after = a':(after tc)}


-- runMachine receives a TM and an input.
-- It returns the final configuration,
-- if the machine stops in an accepting state.

runMachine tm input =
 let startconfig = TMConfig {current = head input,
                             before = [],
                             after = tail input,
                             currState = (start tm)}
     go tc = case oneStep tm tc of
             Nothing -> tc
             Just tc' -> go tc'
 in go startconfig

-- tmEncode receives a TM and an input an returns True,
-- if the TM accepts the input
tmEncode tm input = case runMachine tm input of
                   (TMConfig _ _ _ _) -> True


-- ------------------------------------------------------------
-- Example

-- ex1 TM that searches the last symbol of the input
ex1 =
 let
  d (1,Just 0) = (1,Just 0, MRight)
  d (1,Just 1) = (1,Just 1,  MRight)
  d (1,Nothing) = (2,Nothing, MLeft)
  d (2,_) = undefined
  input = [Just 0,Just 1,Just 0, Nothing]
  tm =
   TM {
    delta = d,
    accepting = [2],
    start = 1
   }
 in runMachine tm input
```

References

Ariola, Z. M. & Felleisen, M. (**1997**). The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M., & Wadler, P. (**1995**). The call-by-need lambda calculus. In *POPL '95: Proceedings of the 22th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 233–246. ACM Press, New York, NY, USA.

Baader, F. & Nipkow, T. (**1998**). *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA.

Barendregt, H. P. (**1984**). *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam, New York.

Church, A. (**1941**). *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey.

Damas, L. & Milner, R. (**1982**). Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, New York, NY, USA.

Gunter, C. A. (**1992**). *Semantics of programming languages: structures and techniques*. MIT Press, Cambridge, MA, USA.

Hankin, C. (**2004**). *An introduction to lambda calculi for computer scientists*. Number 2 in Texts in Computing. King's College Publications, London, UK.

Henglein, F. (**1993**). Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289.

Hindley, J. R. (**1969**). The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60.

Hoare, C. A. R. (**1969**). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (**2006**). *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Kfoury, A. J., Tiuryn, J., & Urzyczyn, P. (**1990**a). Ml typability is dexptime-complete. In *CAAP '90: Proceedings of the fifteenth colloquium on CAAP'90*, pages 206–220. Springer-Verlag New York, Inc., New York, NY, USA.

Kfoury, A. J., Tiuryn, J., & Urzyczyn, P. (**1990**b). The undecidability of the semi-unification problem. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 468–476. ACM, New York, NY, USA.

Kfoury, A. J., Tiuryn, J., & Urzyczyn, P. (**1993**). Type reconstruction in the presence of

polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311.

Kutzner, A. (**2000**). *Ein nichtdeterministischer call-by-need Lambda-Kalkül mit erratic choice: Operationale Semantik, Programmtransformationen und Anwendungen*. Dissertation, J. W. Goethe-Universität Frankfurt. In german.

Kutzner, A. & Schmidt-Schauß, M. (**1998**). A nondeterministic call-by-need lambda calculus. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 324–335. ACM Press.

Mairson, H. G. (**1990**). Deciding ml typability is complete for deterministic exponential time. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 382–401. ACM, New York, NY, USA.

Mann, M. (**2005**a). *A Non-Deterministic Call-by-Need Lambda Calculus: Proving Similarity a Precongruence by an Extension of Howe's Method to Sharing*. Dissertation, J. W. Goethe-Universität, Frankfurt.

Mann, M. (**2005**b). Congruence of bisimulation in a non-deterministic call-by-need lambda calculus. *Electronic Notes in Theoretical Computer Science*, 128(1):81–101.

Maraist, J., Odersky, M., & Wadler, P. (**1998**). The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317.

Milner, R. (**1978**). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

Mitchell, J. C. (**1996**). *Foundations of programming languages*. MIT Press, Cambridge, MA, USA.

Morris, J. H., Jr. (**1968**). *Lambda Calculus Models of Programming Languages*. Ph.D. thesis, MIT, Cambridge, MA.

Mycroft, A. (**1984**). Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 217–228. Springer-Verlag, London, UK.

Peyton Jones, S. L. (**1987**). *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Plotkin, G. D. (**1975**). Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159.

Schmidt, D. A. (**1986**). *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA.

Schmidt-Schauß, M. (**2009**). Skript zur Vorlesung "'Einführung in die Funktionale Programmierung"', WS 2009/10.

Schmidt-Schauß, M. (**2013**). Lecture notes to the lecture "'semantik"' (in german), summer 2013.

Schmidt-Schauß, M. & Sabel, D. (**2010**). On generic context lemmas for higher-order calculi with sharing. *Theoretical Computer Science*, 411(11-13):1521 – 1541.

Schmidt-Schauß, M., Sabel, D., & Machkasova, E. (**2010**). Simulation in the call-by-need lambda-calculus with letrec. In C. Lynch, editor, *Proceedings of the 21st Inter-*

national Conference on Rewriting Techniques and Applications, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 295–310. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

Schmidt-Schauß, M., Schütz, M., & Sabel, D. (**2008**). Safety of Nöcker's strictness analysis. *Journal of Functional Programming*, 18(4):503–551.

Schöning, U. (**2008**). *Theoretische Informatik – kurz gefasst*. Spektrum Akademischer Verlag, 5. Auflage edition.

Sipser, M. (**2013**). *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition.

Stoy, J. E. (**1977**). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA.

Stump, A. (**2013**). *Programming Language Foundations*. Wiley.

Tennent, R. D. (**1994**). *Denotational semantics*, pages 169–322. Oxford University Press, Oxford, UK.

Winskel, G. (**1993**). *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA.