Saarland University
Faculty of Natural Sciences and Technology I

Bachelor'sThesis

A Formal and Constructive Theory of Computation

Author:
Yannick Forster

Advisor:
Prof. Dr. Gert Smolka

Reviewers:
Prof. Dr. Gert Smolka
Prof. Dr. Markus Bläser

Submitted: 11th December 2014

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath:

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent:

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 11th December, 2014

Abstract

This thesis presents a formal development of basic computability theory in constructive type theory. The entire development is carried out in the proof assistant Coq. No classical assumptions are needed for the development.

We base our theory on a minimal functional programming language obtained as a variant of the weak call-by-value lambda calculus. The choice of the programming language is crucial since in a formal development of computability theory complex constructions must be programmed and verified that are merely sketched in standard textbooks.

We show basic undecidability results including the theorems of Scott and Rice. This is the easy part of the formal development. The construction and verification of a step-indexed self-interpreter (universal program) and a dovetailing self-interpreter is considerably more involved. We show that a logically decidable problem is computationally decidable if it is acceptable and co-acceptable. We also show that a problem is acceptable if and only if it is enumerable. Finally, we prove that termination for all arguments is neither acceptable nor co-acceptable.

Acknowledgements

I am very grateful to my advisor Prof. Smolka for offering me this thesis. His support and guidance during the last year, not only in matters of this thesis, were far above of what can be expected.

Then, I want to thank my family, friends and fellow students for their support. Especially I want to thank Kathrin and Felix for proofreading the thesis.

I would also like to thank Chad for a fruitful discussion over the thesis and Prof. Bläser for reviewing it.

The last and biggest thanks go to Saskia. Thank you for your support in all matters. There would have been no way for me to complete this thesis without you.

Contents

Chapter 1

Introduction

Computability theory is one of the oldest fields of computer science. It usually is taught and carried out using classical mathematical methods, see for instance [10] or [11]. We feel that a constructive treatment of computability theory reflects its character in a more genuine way since all basic constructions go through with only small refinements. In this thesis we present a formal and constructive approach to computability theory.

Essential for a compact formalization is the choice of the programming language. We base our studies on a minimal functional programming language, called L. As a compositional language it simplifies the reasoning about programs. L is a subset of the full λ-calculus [2] which was the language the first undecidabilty results were ever published in by Church [4]. It can be seen as a weak call-by-value λ-calculus, similar to the systems considered by Niehren [13], Dal Lago and Martini [7] and Plotkin [16].

The computational objects of L are procedures, which are closed abstractions. Procedures can be executed by applying them to other procedures, they can converge with procedures as values or diverge.

One can represent the usual data types like natural numbers and booleans as procedures via Scott's encoding [6]. Using the same approach it is also easy to implement a self-representation in L, which was done for full λ-calculus by Mogensen [12]. The translation of first-order recursive specifications for total programs into L is routine.

The basis for our formalization is constructive type theory, as implemented in the proof assistant Coq [5]. In Coq every function is total. One can use a technique called step-indexing to model potentially diverging functions in Coq with an additional parameter that bounds the recursion depth. An example for such a function is a step-indexed interpreter for L.

Translating potentially non-total programs into L is achieved by first totalizing the program using a bound for the recursion depth, verifying it and then getting rid of this bound in a general way again. The verification of all programs is based on the equivalence closure of the reduction relation.

Using the mentioned techniques of internalizing functions into procedures and self-representing terms as procedures via Scott's encoding the verification of a self-interpreter for L is straightforward. Starting with this self-interpreter one can implement a *parallel or* construction, which evaluates two terms in parallel by interleaving the evaluation up to fixed reduction lengths stepwise.

Using the self-interpreter and the parallel or we prove various constructions of computability theory. We call the theory developed in this thesis *Constructive Computability Theory (CCT)*. Note that for a problem to be considered decidable in this theory an explicit decider in L needs to be given and proven correct in constructive type theory. Every statement made about decidability in CCT needs to be considered with respect to those two systems, namely L as a model of computation and constructive type theory as a logical framework. By Church's thesis we know that replacing L by another Turing-complete language would not change the results. Using a logical system that is stronger would also preserve the results, but might provide for further results. By basing our studies on constructive type theory we hope to get a more detailed insight into the notions of computation.

All undecidability proofs reduce to the undecidability of the self-halting problem, which was discovered by Turing [18] and Church [4]. In order to show the undecidability of a problem one assumes a decider and constructs a decider for the self-halting problem by explicitly giving and verifying such a program.

Scott's theorem provides sufficient criteria for the undecidability of a problem. It was first published by Barendregt [2] for full λ-calculus. We adapt the proofs to L and additionally give proofs of the underlying fixed-point theorems.

We also show Rice's theorem [17], where the textbook proof needs slight changes to work constructively.

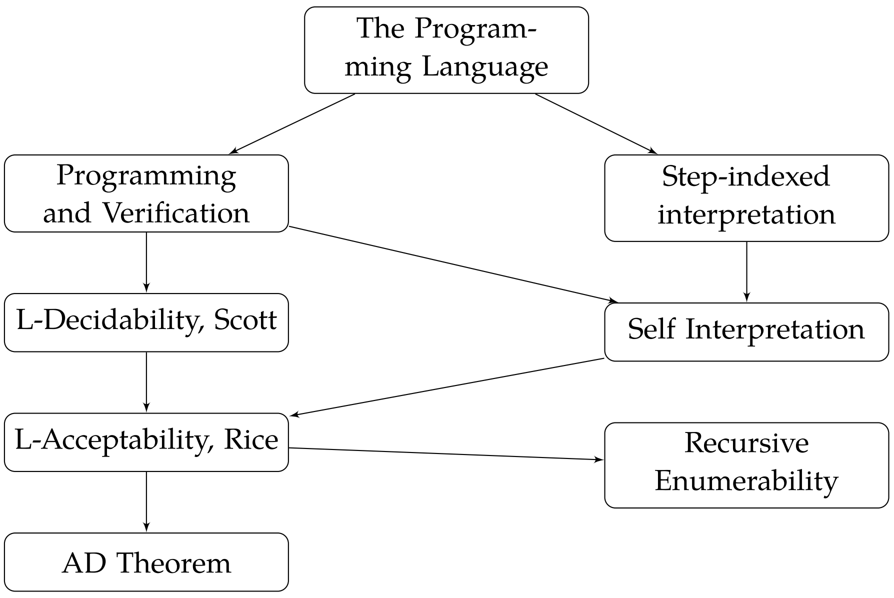
We show that for a propositionally decidable problem acceptability together with co-acceptability implies decidability. We also define the notion of enumerability in L and show that a problem is enumerable if and only if it is acceptable. To do so we need an explicitly constructed surjection from natural numbers to the terms of L, which is a good example for the computational power of L.

Related Work

Norrish [15] presents a development of computability theory covering Rice's theorem in classical type theory carried out with the proof assistant HOL4. Norrish employs full λ-calculus as model of computation and shows the equivalence with Kleene's system of recursive functions. Dal Lago and Martini [7] show on paper that L and Turing machines can simulate each other with polynomial overhead. They use Scott's data encoding to represent Turing machines. Asperti and Ricciotti [1] construct and verify a universal Turing machine with the Coq-like proof-assistant Matita. They do not consider a step-indexed universal machine. Xu, Zhang, and Urban [20] construct and verify a universal Turing machine using Isabelle/HOL. They follow the book of Boolos et al. [3] and verify a compiler translating descriptions of recursive functions to Turing machines. They obtain a universal Turing machine from a universal recursive function. They do not consider a step-indexed universal machine.

Contribution

We seem to be the first to give a (formal) development of computability theory in constructive type theory covering acceptable problems and the theorems of Scott and Rice. Obtaining a constructive development was not difficult, we just had to be careful with the formulation of some undecidability results, most notably Rice's theorem. We are the first to base such a development on L rather than full λ-calculus. As it turns out, L simplifies such a development. The weak reduction equivalence of L (procedures are equivalent only if they are identical) suffices for the proof of Scott's theorem and the verification of the internalized functions.



Structure of the thesis

Chapter 2

The Programming Language

Essential for our goal of a compact formalization is the choice of the programming language. We choose L as our programming lanugage, which can be seen as weak call-by-value lambda calculus. It is a slight variation of the system used by Dal Lago and Martini [7] who show that L and Turing machines can simulate each other. L is related to Plotkin's call-by-value calculus [16] and Church's full λ-calculus [4]. It is simpler than full λ-calculus since reductions can only happen on the top level, which is computationally more realistic and leads to unique normal forms.

The reduction in L is uniformly confluent, which was observed by Niehren [13], and has a parametric diamond property.

2.1 Syntax of L

We define L as it will be used throughout this thesis. We employ de Bruijn terms [8] for the formal definition:

$$s, t, u \ ::= \ n \ \mid \ s\ t \ \mid \ \lambda s \qquad (n \in \mathbb{N})$$

Terms of the form n where n is a natural number are called **variables**. Those of the form $s\ t$ **applications** and those of the form λs **abstractions**. The letters v and w will range over abstractions only. We define a substitution for terms that replaces any free occurrence of the variable k in the term s with u.

$$\begin{aligned} n_u^k &= \mathsf{if}\ n = k\ \mathsf{then}\ u\ \mathsf{else}\ n \\ st_u^k &= (s_u^k)(t_u^k) \\ (\lambda s)_u^k &= \lambda(s_u^{Sk}) \end{aligned}$$

We use the notation Sk for the successor of a natural number k. Note that the definition of substitution is not capture-avoiding, but that s_u^0 yields the same result as a beta reduction of $(\lambda s)u$ in full λ-calculus. Our reduction is only well-behaving if u is a closed term and s has no free variables except k. We call a term s **closed** if $\forall ku. s^k_u = s$, i.e. if there is no free variable that could be substituted. We call a term k-closed if k is greater than all free variables of the term.

$$\frac{k > n}{\mathit{closed}_k\ n} \qquad \frac{\mathit{closed}_k\ s \qquad \mathit{closed}_k\ t}{\mathit{closed}_k\ (s\ t)} \qquad \frac{\mathit{closed}_{Sk}\ s}{\mathit{closed}_k\ (\lambda s)}$$

0-closedness and closedness of a term are equivalent:

Fact 2.1 $m \geq k \rightarrow \mathit{closed}_k\ s \rightarrow \mathit{closed}_m\ s$

Fact 2.2 $\mathit{closed}_k\ s \rightarrow s^k_u = s$

Fact 2.3 $(\forall n \geq k. s^n_u = s) \rightarrow \mathit{closed}_k\ s$

Lemma 2.4 $\mathit{closed}_0\ s \leftrightarrow \mathit{closed}\ s$

Proof For the direction from left to right assume that s is 0-closed. We need to prove that $s^k_u = s$ for any natural number k and any term u. Since $k \geq 0$ we find that s is closed because of Facts 2.1 and 2.2.

For the direction from right to left assume that s is closed. To show that it is also 0-closed by Fact 2.3 it is enough to show that for all $n \geq 0$ we have $s^n_u = s$, which is exactly the assumption that s is closed. ∎

Fact 2.5 *Closedness of a term is computationally decidable in Coq.*

If a term is not closed, we call it **open**. A variable is always open. If a term is closed, we call it **combinator**. For a closed abstraction we write **procedure**. We fix some well-known combinators now, which will be used throughout the thesis:

$$I := \lambda x.x \qquad \omega := \lambda x.x\ x$$
$$K := \lambda xy.x \qquad \Omega := \omega\ \omega$$

I is simply the identity, while Kv represents a constant function. Ω is a always diverging combinator.

Note that we did not use de Bruijn indices to define this terms. When it comes to reading, frequent usage of de Bruijn indices can be confusing and hard to understand. Thus we will use usual binder notation with named bound variables in the thesis and use the indices only at the bottom of the underlying formalization. The letters x, y and z will range over variables only. The terms I, K and Ω above would be $\lambda 0$, $\lambda\lambda 1$ and $\lambda 00$ written with indices. If we do not use a bound variable in the following term we might omit it and simply write λs instead of $\lambda x.s$.

We will omit parantheses following the rules $s\ t\ u = (s\ t)\ u$ and $\lambda x.s\ t = \lambda x.(s\ t)$. Combinators will always be written with an uppercase letter.

2.2 Reduction and Equivalence

The dynamic semantics of our calculus is defined via a one-step reduction relation $\succ$. In contrast to Dal Lago and Martini [7] we do not treat variables as values and thus allow β-reduction only if both sides are abstractions.

$$\frac{}{(\lambda s)\ v \succ s^0_v} \qquad \frac{s \succ s'}{s\ t \succ s' t} \qquad \frac{t \succ t'}{st \succ st'}$$

For $s \succ t$ we say s **reduces to** t. In such a case we call t the **successor** of s and s the **predecessor** of t. A term that has a successor is called **reducible**, and a term that has no successor is said to be **normal**. We will use $\succ^*$ for the reflexive transitive closure of $\succ$ and $\succ^k$ for precisely k reduction steps. If $s \succ^* v$ where v is normal we write $s \Downarrow v$ and call v the **normal form** of s. If v is a procedure, then we call v the **value** of s.

Fact 2.6

1. *A combinator is either a procedure or reducible.*
2. *A combinator is normal if and only if it is a procedure.*

We say that a term s **converges** if it has a normal form. In mathematical notation we simply write $s \Downarrow$. A term that has no normal form **diverges**.

Note that the reduction is weak in the sense that no reduction inside an abstraction is possible. It is call-by-value, because only applications where both sides are abstractions (and thus values) can be reduced using the first rule. We call reductions using this rule β-reductions. In every reduction step there is exactly one β-reduction. The other rules will be referred to as left reduction rule and right reduction rule respectively.

If a closed term t gets substituted into a term s where only the variable with index 0 is free during β-reduction, the result is closed again. Thus during a β-reduction concerning two closed terms the result will be closed again. This can be generalized to the following lemma:

Lemma 2.7 $\mathit{closed}_{Sk}\ s \to\ \mathit{closed}_k\ t \to\ \mathit{closed}_k\ s^k_t$

Proof By induction over s and an examination of the first assumption. ∎

Corollary 2.8 *If s is closed and $s \succ t$, then t is closed.*

Proof By induction over $s \succ t$. ∎

To obtain an equational theory for L we need to define the reflexive, transitive, symmetric closure of $\succ$. Dal Lago and Martini [7] do not consider an equivalence closure for their relation. The system studied by Plotkin [16] bases on an equational theory, but is not weak, since it allows reduction below binders. The equivalence closure for $\succ$ will be written as $\equiv$.

$$\frac{s \succ t}{s \equiv t} \qquad \frac{}{s \equiv s} \qquad \frac{s \equiv t}{t \equiv s} \qquad \frac{s \equiv t \quad t \equiv u}{s \equiv u}$$

Fact 2.9 *If $s \equiv t$ then $s \Downarrow\ \leftrightarrow t \Downarrow$.*

Some properties of $\equiv$ ease the formal proofs a lot. Since no reduction order is given, one can rewrite anywhere on top-level in a term:

Fact 2.10 *If $s \equiv s'$ and $t \equiv t'$, then $s\,t \equiv s'\,t'$.*

In order to prove the uniqueness of normal forms equivalent normals forms (which are always abstractions) need to be equal. This is a direct consequence of the weakness of the reduction relation.

Fact 2.11 *If $\lambda s \equiv \lambda t$, then $s = t$.*

2.3 Uniform Confluence

$$\begin{tikzcd} & s \arrow[dl] \arrow[dr] & \\ t_1 \arrow[dr] & & t_2 \arrow[dl] \\ & u & \end{tikzcd}$$

Figure 2.1: Uniform Confluence

L is a uniformly confluent system, which was identified by Niehren [13; 14]. Therefore it is confluent and every reduction path to a normal form has the same length. We essentially follow the proofs from [7].

Theorem 2.12 (Uniform Confluence of *L*) *If $s \succ t_1$ and $s \succ t_2$, then either $t_1 = t_2$ or there is a term u such that $t_1 \succ u$ and $t_2 \succ u$.*

Proof By induction over s. Only for the case where $s = s_1\ s_2$ there is something to prove. We analyze the rule $s \succ t_1$ used:

- If $s \succ t_1$ via β-reduction, then β-reduction was the only possible rule for any reduction $s \succ t$ and $s \succ t_2$ used this rule also. Thus $t_1 = t_2$.
- If $s \succ t_1$ via the left reduction rule, then $s \succ t_2$ could have used either the same rule (then $t_1 = t_2$) or the right-reduction rule, in which case there is u such that $t_1 \succ u$ via the right reduction rule and $t_2 \succ u$ via the left reduction rule. Note that $s \succ t_2$ could not have been a β-reduction then.
- If $s \succ t_1$ via the right-reduction rule, then the case is symmetric to the one above.

■

$$\begin{tikzcd} & & s \arrow[dl] \arrow[dr] & \\ & v \arrow[dl, "m"'] & & t_2 \\ t_1 & & & \end{tikzcd}$$

$$\begin{tikzcd} & & s \arrow[dl] \arrow[dr] & \\ & v \arrow[dl, "m"'] \arrow[dr] & & t_2 \arrow[dl] \\ t_1 \arrow[dr, "k"'] & \text{by IH} & u \arrow[dl, "l"] & \\ & u' & & \end{tikzcd}$$

Figure 2.2: 2nd case of parametric semi-confluence

Theorem 2.13 (Parametric Diamond Property) *If $\succ$ is any uniformly confluent relation, then: if $s \succ^m t_1$ and $s \succ^n t_2$, then either $t_1 = t_2$ or there is a term u and natural numbers $k \le n$, $l \le m$ such that $t_1 \succ^k u$ and $t_2 \succ^l u$ and $m + k = n + l$.*

Proof We begin with the following Lemma:

Lemma 2.14 (Parametric Semi Confluence) *If $s \succ^m t_1$ and $s \succ t_2$, then there is a term u and natural numbers $k \le 1$, $l \le m$ such that $t_1 \succ^k u$ and $t_2 \succ^l u$ and $m + k = 1 + l$.*

Proof By an induction over m:

- If $m = 0$, then $s = t_1$ and one obviously can choose $k = 1$, $l = 0$ and $u = t_2$.
- If $m = Sm$, then the situation is as on the left in Figure 2.2. We use the property that $\succ$ is uniformly confluent. That is, either $v = t_2$, where we can choose $k = 0$, $l = m$ and $u = t_1$. Or we have u where $v \succ u$ and $t_2 \succ u$. By using the inductive hypothesis we are now able to find a u' to choose for u together with k and l and can choose $k = k$ and $l = Sl$, so that we are done. This case is shown on the right in Figure 2.2. ■

Using this parametric semi-confluence property we are now able to prove the Parametric Diamond Property again by induction over m. In any case we need to give suitable k, l and u.

- $m = 0$: Then $s = t_1$ and we can choose $k = n$, $l = 0$ and $u = t_2$.
- $m = Sm$: We have a situation similar to Figure 2.2. By using the parametric semi-confluence we can find k, l and u. By using the inductive hypothesis we get l', k' and u', can fill the diagram as in Figure 2.2 and choose $k = l'$, $l = k + k'$ and $u = u'$. ■

Corollary 2.15 *L has the parametric diamond property.*

Corollary 2.16 *L is confluent.*

Corollary 2.17 (Church-Rosser property for *L*) *If $s \equiv t$, then $s \succ^* u$ and $t \succ^* u$ for some term u.*

Corollary 2.18 *If $s \equiv v$ where v is an abstraction, then $s \succ^* v$.*

The weakness of the reduction combined with confluence ensures uniqueness of normal forms.

Fact 2.19 *If $s \Downarrow v_1$ and $s \Downarrow v_2$, then $v_1 = v_2$.*

Chapter 3

Programming and Verification

We now turn towards programming in L. We need the ability to define combinators via recursive specifications and a way to represent the usual known data types, such as booleans and natural numbers.

Another essential aspect of computability theory is self-representation. We need an elegant way how terms of L can represent other terms. We will internalize the type *term* as well as booleans and natural numbers using Scott's encoding [6].

Jansen [9] gives an introduction to programming in full λ-calculus using Scott's encoding.

3.1 Recursion

To implement recursive specifications we need a fixed-point combinator R now, that is a combinator such that for any procedure f the equation $R\ f \equiv f(R\ f)$ holds. In our call-by-value setting this is not possible.

Thus we define the call-by-value version of a fixed-point combinator found by Turing [19] following Dal Lago and Martini [7]:

$$A := \lambda zf.f(\lambda x.zzfx)$$
$$R := A\ A$$

Fact 3.1 *If s is a procedure, then $Rs \succ^2 s(\lambda x.Rsx)$.*

Unlike in full λ-calculus a recursive combinator $R\ s$ is normalizing. It is not possible to give a normalizing recursive combinator for full λ-calculus. Note that the η-expansion of Rs to $\lambda x.Rsx$ is not problematic, since for a recursive call the arguments are always abstractions (or at least reduce to one) and it holds that:

Fact 3.2 (η-Equivalence) *If s is a combinator and v an abstraction, then $(\lambda x.s\,x)\ v \equiv s\,v$.*

3.2 Datatypes

Beside recursion the other key part of programming are data types. The most well-known encoding of data types into terms of a λ-calculus are Church's encoding (mostly used for Church numerals, see [2] [4]) and the representation of a term as the Church numeral of its Gödel-number [4]. In a call-by-value calculus, Church encodings are no option. Thus we base all our programs on Scott's encoding [6], which was used for terms by Mogensen [12]. In combination with the combinator R we obtain a powerful programming language which allows a direct translation of almost all data types using *match* and *fix* constructs.

3.2.1 Booleans and Natural Numbers

As a first example we present how to encode booleans and natural numbers in L:

$$\begin{aligned} \textit{true} &:= \lambda xy.x \\ \textit{false} &:= \lambda xy.y \\ \overline{0} &:= \lambda zs.z \\ \overline{Sn} &:= \lambda zs.s\,\overline{n} \\ \textit{Succ} &:= \lambda xzs.s\ x \\ \textit{Succ}\,\overline{n} &\equiv \overline{Sn} \end{aligned}$$

The last equation is the correctness statement for *Succ* and easily proven by induction over n.

Fact 3.3 *true, false and $\overline{n}$ are procedures.*

For the usage of booleans the following two combinators realizing *boolean and* and *boolean or* are helpful. Note that both arguments need to be evaluated until the result can be computed, which means that even if one argument is *true*, but the other one diverges, the boolean or of them will diverge. An *if-then-else* construct can be realized by a simple application of the if- and else-branch to the boolean which should be checked.

$$\begin{aligned} \textit{andalso} &:= \lambda ab.a\ b\,\textit{false} \\ \textit{orelse} &:= \lambda ab.a\ \textit{true}\ b \end{aligned}$$

We are now able to define a combinator which adds two encoded numbers:

$$\textit{Add} := R\ (\lambda amn.n\,\overline{0}\ (\lambda m'.\textit{Succ}\ (a\ m'\ n)))$$

One can see almost all phenonema concerning the definition of combinators in L at this example. First if one wants do define a recursive combinator F it will always be $F := R(\lambda f.\ \dots)$ where the variable f can be used to make a recursive call. Second, the match over a dataterm is implemented using simple applications. The fragment **match** n **with** 0 ⇒ s | S n ⇒ t in Coq is translated to L with $n\ s\ (\lambda n.t)$.

We are now able to verify the correctness of *Add*. To do this we will always first verify that the recursive equations hold, in this case:

$$\begin{aligned} \mathit{Add}\,\overline{0}\,\overline{n} &\equiv \overline{n} \\ \mathit{Add}\,\overline{Sm}\,\overline{n} &\equiv \mathit{Succ}\ ((\lambda x.\mathit{Add}\ x)\ \overline{m}\,\overline{n}) \end{aligned}$$

We then can prove the correctness statement for *Add*:

Lemma 3.4 (Correctness of *Add*) $\mathit{Add}\,\overline{m}\,\overline{n} \equiv \overline{m+n}$

Proof By an induction over m this boils down to the two Dedekind equations:

$$\begin{aligned} \mathit{Add}\,\overline{0}\,\overline{n} &\equiv \overline{n} \\ \mathit{Add}\,\overline{Sm}\,\overline{n} &\equiv \mathit{Succ}\,\overline{m+n} \end{aligned}$$

The first equivalence directly follows from the recursive specification. The second follows from the fact that $\overline{m}$ is a procedure, the recursive specification and the inductive hypothesis:

$$\begin{aligned} \mathit{Add}\,\overline{Sm}\,\overline{n} &\equiv \mathit{Succ}\ ((\lambda x.\mathit{Add}\ x)\ \overline{m}\,\overline{n}) && \text{(rec. spec.)} \\ &\equiv \mathit{Succ}\ (\mathit{Add}\,\overline{m}\,\overline{n}) && (\overline{m}\text{ is a value}) \\ &\equiv \mathit{Succ}\,\overline{m+n} && \text{(IH)} \\ &\equiv \overline{S(m+n)} && \text{(Correctness of } \mathit{Succ}) \end{aligned}$$

■

An important property of the encoding for natural numbers is injectivity:

Lemma 3.5 $\overline{n} \equiv \overline{m} \rightarrow\ n = m.$

Proof Because $\overline{m}$ and $\overline{n}$ are both procedures we know that $\overline{m} = \overline{n}$. The claim then follows by induction over n. ■

3.2.2 Terms

To internalize a universal self-interpreter for L we need a way to represent terms. As with booleans and natural numbers the standard strategy for the translation of data types into Scott encodings applies. This strategy is described in detail in Section 3.3.

$$\begin{aligned}\ulcorner n \urcorner &:= \lambda v\, a\, l. v n\\ \ulcorner s\, t \urcorner &:= \lambda v\, a\, l. a \ulcorner s \urcorner \ulcorner t \urcorner\\ \ulcorner \lambda s \urcorner &:= \lambda v\, a\, l. l \ulcorner s \urcorner\end{aligned}$$

We read $\ulcorner s \urcorner$ as **quote** s and refer to terms of the form $\ulcorner s \urcorner$ as **data terms**.

Fact 3.6 *Every data term is a procedure.*

We know that the function $\ulcorner\ .\ \urcorner$ is injective:

Lemma 3.7 $\ulcorner t \urcorner \equiv \ulcorner s \urcorner \rightarrow\ s = t.$

Proof Follows by $\ulcorner s \urcorner = \ulcorner t \urcorner$ and induction over s. ■

We can internalize the three constructors directly

$$\begin{aligned}\mathit{Var} &:= \lambda x. \lambda v\, a\, l. v\ x\\ \mathit{App} &:= \lambda x y. \lambda v\, a\, l. a\ x\ y\\ \mathit{Lam} &:= \lambda x. \lambda v\, a\, l. l\ x\end{aligned}$$

and obtain the following equivalences:

$$\begin{aligned}\mathit{Var}\, \overline{n} &\equiv \ulcorner n \urcorner\\ \mathit{App} \ulcorner s \urcorner \ulcorner t \urcorner &\equiv \ulcorner s\, t \urcorner\\ \mathit{Lam} \ulcorner s \urcorner &\equiv \ulcorner \lambda s \urcorner\end{aligned}$$

As an example of programming with terms we will define and internalize the size of a term. The size of a term is precisely the number of constructors involved:

$$\begin{aligned}|n| &= 1\\ |s\, t| &= 1 + |s| + |t|\\ |\lambda s| &= 1 + |s|\end{aligned}$$

We then can define a recursive combinator *Size* fulfilling the following specification:

$$\begin{aligned}\mathit{Size} \ulcorner n \urcorner &\equiv \overline{n}\\ \mathit{Size} \ulcorner s\, t \urcorner &\equiv \mathit{Add}\ \overline{1}\ (\mathit{Add}\ ((\lambda x. \mathit{Size}\ x) \ulcorner s \urcorner)((\lambda x. \mathit{Size}\ x) \ulcorner t \urcorner))\\ \mathit{Size} \ulcorner \lambda s \urcorner &\equiv \mathit{Add}\ \overline{1}\ ((\lambda x. \mathit{Size}\ x) \ulcorner s \urcorner)\end{aligned}$$

This is the last example where we give the exact definition of the term in L. From now on we assume that every such trivial specification can be translated using the usual methods to L. In Section 3.3 we give an overview how to generically transfer every recursive specification into a combinator.

$$Size := R(\lambda f\,t.t(\lambda n.\ \overline{1})(\lambda t_1 t_2.Add\ (f\,t_1)\ (f\,t_2))(\lambda t.Add\,\overline{1}\ (f\,t))$$

Again we prove that *Size* correctly internalizes the $|\,.\,|$-function:

Lemma 3.8 (Correctness of *Size*) *Size* $\ulcorner s \urcorner \equiv \overline{|s|}$

Proof By induction over s. Every case follows directly from the inductive hypotheses, the correctness of *Add* and η-equivalence. ■

3.2.3 Pairs of Natural Numbers

The internalization of pairs of natural numbers is straightforward. We define:

$$\overline{(n,m)} := \lambda p.p\,\overline{n}\,\overline{m}$$

and

$$Pair := \lambda n\ m\ p.p\ n\ m$$

such that we have

$$Pair\ \overline{n}\ \overline{m} \equiv \overline{(n,m)}$$

3.3 Verification of Internalized Functions

Internalizing a Coq-function and verifying its correctness is routine. We first define what internalization precisely means:

Definition 3.9 *A combinator u internalizes a function f : term $\rightarrow$ term if for every term s:*

$$u \ulcorner s \urcorner \equiv \ulcorner f\ s \urcorner$$

We adapt this definition for functions of arbitrary type $X \rightarrow Y$ by specifying how to encode any data type into *term*. We will always use Scott's encoding for this purpose

Assume that X has n constructors $c_1, \ldots, c_n$. For any k-ary constructor c_i an element $c_i\ x_1 \ldots x_k$ is represented as

$$\lambda\, c_1 \ldots\ c_n.\ c_i\ x_1 \ldots x_k$$

We sketch this approach for the type of lists, which will be reused in Section 9.4. Lists over a type Z have two constructors, namely *nil* and *cons* : $Z \to listZ \to listZ$. nil has arity 0 and is represented by the term $\lambda nc.n$. The constructor *cons* is 2-ary. A term *cons* x l can thus be represented as $\lambda nc.c\ x'\ l'$ where x' and l' are the representations for x and l respectively. The list $[1, 2] =$ *cons* 1 $($*cons* 2 $nil)$ would be represented as $\lambda nc.c\ \overline{1}\ (\lambda nc.c\ \overline{2}\ (\lambda nc.n))$

Now every term $\lambda\ c_1 \dots\ c_n.\ c_i\ x_1 \dots x_k$ yields a match construct. The Coq-match

```
match t with
    | $c_1 x_1 \dots x_{k_1} \Rightarrow f_1\ x_1 \dots x_{k_1}$
    | ...
    | $c_n x_1 \dots x_{k_n} \Rightarrow f_n\ x_1 \dots x_{k_n}$
end
```

is simply done with $t\ f_1\ \dots\ f_n$. So if we want to program the *isnil* function with isnil = **fun** a ⇒ **match** a **with** nil ⇒ true | cons c l ⇒ false **end** we write

$$isnil = \lambda a.a\ true\ (\lambda cl.false)$$

There is one case where this pattern needs to be extended. For a recursive call in a case where the constructor is 0-ary, one needs to η-expand all terms. Consider for instance the case where the program recurses only in the nil-case or in the 0-case. Before the match, both alternatives need to be evaluated, so to make the recursion only happen if the particular case is chosen, one needs to put an additional binder over all cases and apply the result to an arbitrary procedure. This technique is called lambda-lifting. The generic match is thus translated to $t\ (\lambda f_1)\ \dots\ (\lambda f_n)\ I$. We could use this translation always, since it would never introduce errors, but if possible we stick with the first version, which is easier to read.

The verification of the procedure is also almost mechanical. One first verifies that the recursive equations hold up to η-equivalence. Then the correctness statement (see Definition 3.9) is proven step for step.

The rule is: If the function matches over a term t without recursing over it, one simply makes a case analysis over the structure of the term. If the function recurses over a term t after matching over it, one simply proceeds with an induction over t.

Chapter 4

Decidability and Scott's Theorem

4.1 Decidable Predicates

Classical computability theory talks about the decidability of problems. Because we are working in a type theoretic setting choosing predicates to represent problems seems natural. A predicate will always be of type $\mathbf{T} \rightarrow$ **Prop**, where $\mathbf{T}$ is shorthand for *term*. The letters P and Q range over predicates only.

We define predicates P with the notation $\boldsymbol{\lambda}t.\ Q\ t$ which means $P\ t \leftrightarrow Q\ t$. We define $\mathbf{P}$ as $\boldsymbol{\lambda}t.$ *t is a procedure* and $\mathbf{C}$ as $\boldsymbol{\lambda}t.$ *t is a combinator*. We use the notation $P \subseteq Q$ to express that $\forall t. P\ t \rightarrow Q\ t$, so $P \subseteq \mathbf{P}$ means that every term satisfying P is a procedure. We will write $\overline{P}$ for the complementary predicate $\boldsymbol{\lambda}t.\ \neg P\ t$.

There are several notions of decidability. First, there is Turing decidability, which refers to the general term of decidability in any programming language. Second, we define what it means to be decidable in Constructive Type Theory. For convenience, we refer to this as Coq-decidable, but the restriction to Coq as the particular implementation is not relevant.

Definition 4.1 *A predicate $P : \mathbf{T} \rightarrow$ **Prop** is called **Coq-decidable** if there is a function of type $\forall t : \mathbf{T}. \{P\ t\} + \{\neg P\ t\}$.*

We also define what it means for a predicate to be L-decidable.

Definition 4.2 *A predicate P is **L-decidable** if there is a procedure u such that:*

$$\forall t : \mathbf{T}. u \ulcorner t \urcorner \equiv \mathit{true} \wedge P\ t \vee u \ulcorner t \urcorner \equiv \mathit{false} \wedge \neg P\ t$$

*We then call say that u is a **decider for** P. Any predicate for which we can prove that it is not L-decidable is called **L-undecidable**.*

Note that every L-decidable predicate is also Turing decidable. An L-undecidable predicate is Turing undecidable.

4.2 The Self-Halting Problem

The self-halting problem is often attributed to Alan Turing who mentions it in [18], but got independently discovered at least by Alonzo Church in [4]. The general concern of the problem is to determine if a given term of a formal system will converge in its evaluation or diverge.

Lemma 4.3 (Undecidability of the Self-Halting Problem) *The self-halting problem* $\boldsymbol{\lambda} t.\ t \ulcorner t \urcorner \Downarrow$ *is L-undecidable.*

Proof Assume that there is a term u deciding the self-halting problem, that is $\forall t. u \ulcorner t \urcorner \equiv \mathit{true} \wedge t \Downarrow \vee u \ulcorner t \urcorner \equiv \mathit{false} \wedge \neg(t \Downarrow)$.

Now define $t := \lambda x.u\ x\ (\lambda\Omega)(\lambda I)I$.

- Assume $u \ulcorner t \urcorner \equiv \mathit{true}$ and $t \ulcorner t \urcorner \Downarrow$. We have

$$\begin{aligned} t \ulcorner t \urcorner &\equiv u \ulcorner t \urcorner\ (\lambda\Omega)\ (\lambda I)\ I \\ &\equiv \mathit{true}\ (\lambda\Omega)\ (\lambda I)\ I \\ &\equiv (\lambda\Omega)\ I \\ &\equiv \Omega \end{aligned}$$

  But Ω diverges, while $t \ulcorner t \urcorner \Downarrow$ by assumption. Since $t \ulcorner t \urcorner \equiv \Omega$, this is a contradiction, because convergence is closed under $\equiv$. To make the contradiction clear, note that then $t \ulcorner t \urcorner \Downarrow \wedge \neg(t \ulcorner t \urcorner \Downarrow)$ would hold.

- Assume $u \ulcorner t \urcorner \equiv \mathit{false}$ and $t \ulcorner t \urcorner \Uparrow$. We have

$$\begin{aligned} t \ulcorner t \urcorner &\equiv u \ulcorner t \urcorner\ (\lambda\Omega)\ (\lambda I)\ I \\ &\equiv \mathit{false}\ (\lambda\Omega)\ (\lambda I)\ I \\ &\equiv (\lambda I)\ I \\ &\equiv I \end{aligned}$$

  But I converges, while $t \ulcorner t \urcorner \Uparrow$ by assumption. Since $t \ulcorner t \urcorner \equiv I$, this is again a contradiction.

Thus, in any case we get a contradiction and there can not be a decider for the self-halting problem. ∎

4.3 Fixed Point Theorems and Scott's Theorem

We have already shown that there are undecidable problems. We are now interested in more general results and criteria for predicates that are sufficient for their undecidability.

We show **Scott's Theorem**, proven in [2] as Theorem 6.6.2 for full λ-calculus. It follows from a Fixed Point Theorem, which is interesting by its own means and shown below.

4.3.1 Fixed Point Theorems

Both following proofs can be found in [2] for full λ-calculus (Theorems 2.1.5 and 6.5.9). We adapt Barendregt's proofs to L, which essentially means that we use Scott's encoding instead of Gödel-Church encoding.

Theorem 4.4 (First Fixed Point) *For every combinator s there exists a combinator t such that $s\ t \equiv t$.*

Proof Define $\omega_s := \lambda x.s\ (x\ x)$ and $F := \omega_s\ \omega_s$. Now $F = (\lambda x.s\ (x\ x))(\lambda x.s\ (x\ x)) \succ s\ ((\lambda x.s\ (x\ x))(\lambda x.s\ (x\ x))) = s\ (\omega_s\ \omega_s) = s\ F$. ■

Theorem 4.5 (Second Fixed Point) *For every combinator s there exists a combinator t such that $s \ulcorner t \urcorner \equiv t$.*

Proof Define $A := \lambda x.s(\mathit{App}\, x\, (Q\, x))$ and $t := A \ulcorner A \urcorner$. Then $t \succ s\ (\mathit{App} \ulcorner A \urcorner (Q \ulcorner A \urcorner))$ $\equiv s\ (\mathit{App} \ulcorner A \urcorner \ulcorner\ulcorner A \urcorner\urcorner) \equiv s \ulcorner A \ulcorner A \urcorner\urcorner \equiv s \ulcorner t \urcorner$. ■

Corollary 4.6 *There is a combinator t with $t \equiv \ulcorner t \urcorner$.*

4.3.2 Scott's Theorem

We essentially follow [2] and adapt the proofs to Scott's encoding.

Theorem 4.7 (Scott) *A predicate P is L-undecidable if it satisfies the following conditions:*

1. *P is only satisfied by combinators: $P \subseteq \mathbf{C}$.*
2. *P is closed under reduction equivalence: For equivalent combinators s and t it holds that $P s \rightarrow P\ t$.*
3. *P is nontrivial: There are combinators s_1 and s_2 with $P s_1$ and $\neg(P s_2)$.*

Proof Let $P \subseteq \mathbf{C}$ be as a nontrivial predicate closed under reduction equivalence where $P s_1$ holds and $P s_2$ does not for combinators s_1 and s_2. Assume u were a decider for P. Now define the combinator

$$s := \lambda x.u\ x\ (\lambda s_2)\ (\lambda s_1)\ I$$

By Theorem 4.5 we know that we have a combinator t with

$$t \equiv s \ulcorner t \urcorner \equiv u \ulcorner t \urcorner\ (\lambda s_2)\ (\lambda s_1)\ I$$

Because u is a decider for P we have:

- Either $u \ulcorner t \urcorner \equiv$ *true*, then Pt holds and $t \equiv s_2$, but $\neg P s_2$. Contradiction.
- Or $u \ulcorner t \urcorner \equiv$ *false*, then $\neg Pt$ holds and $t \equiv s_1$, but $P s_1$. Contradiction. ∎

Note that the proof works exactly analogous to Lemma 4.3. It extracts the properties that make the self-halting problem L-undecidable, but needs one application of the second fixed point theorem, where the original proof could just use self-application.

Corollary 4.8 $\boldsymbol{\lambda} t.\ \mathbf{C} t \wedge t \Downarrow$ *is L-undecidable.*

Proof Convergence is nontrivial and closed under equivalence. ∎

Lemma 4.9 *For every combinator t the predicate* $\boldsymbol{\lambda} s.\ \mathbf{C} s \wedge s \equiv t$ *is L-undecidable.*

Proof Assume we had a decider u for $\boldsymbol{\lambda} s.\ \mathbf{C} s \wedge s \equiv t$. We get a contradiction by applying Scott's Theorem, which shows that the predicate is undecidable.

First, the predicate is only satisfied by combinators and respects term equivalence.

The interesting part is to show nontriviality:

1. We need to find a combinator that is equivalent to t, where we can simply pick t.
2. We then need to give a combinator not equivalent to t. Here we first use u to decide if $I \equiv t$. If this is the case, we choose Ω, since $I \not\equiv \Omega$. If $I \not\equiv t$, then we can choose I. ∎

Corollary 4.10 *For every procedure t the predicate* $\boldsymbol{\lambda} s.\ s \equiv t$ *is L-undecidable.*

Note that the proof of 4.9 heavily relies on the fact that we first assumed a decider and then showed that the predicate is L-undecidable. Directly invoking Scott's Theorem would not work, since we had no possibility to find a term that is not equivalent to t.

We show that the equivalence relation of L is L-undecidable. Because our predicates are restricted to $\mathbf{T}$, we can not work on pairs of terms. Thus we encode the pair (s, t) as $\ulcorner s\ t \urcorner$:

Theorem 4.11 $\boldsymbol{\lambda} s.\ \exists s_1 s_2 . s = \ulcorner s_1\ s_2 \urcorner \wedge s_1 \equiv s_2$ *is L-undecidable.*

Proof Assume u were a decider. Then the equivalence to the combinator I is L-decidable, which contradicts Lemma 4.9. The following is a decider for the predicate $P := \boldsymbol{\lambda} s.\ s \equiv I$:

$$v := \lambda x. u\ (Q\ (App\ x \ulcorner I \urcorner))$$

Let s be a combinator. We use u to decide if Ps holds:

1. In the first case we get terms s' and t' such that $\ulcorner s\ I \urcorner = \ulcorner s'\ t' \urcorner$, $s' \equiv t'$ and $u \ulcorner\ulcorner s\ I \urcorner\urcorner \equiv \mathit{true}$.

   We want to show that Ps holds and $v \ulcorner s \urcorner \equiv \mathit{true}$:

   For this we need to show that $s \equiv I$, which follows because $\ulcorner s\ I \urcorner = \ulcorner s'\ t' \urcorner$ implies $s = s'$ and $I = t'$ by Lemma 3.7 and thus $s \equiv s' \equiv t' \equiv I$.

   Also, $v \ulcorner s \urcorner \equiv u\ (Q\ (\mathit{App} \ulcorner s \urcorner \ulcorner I \urcorner)) \equiv u \ulcorner\ulcorner s\ I \urcorner\urcorner \equiv \mathit{true}$.

2. In the second case we know $\nexists s't'. \ulcorner sI \urcorner = \ulcorner s't' \urcorner \wedge s' \equiv t'$ and $u \ulcorner\ulcorner s\ I \urcorner\urcorner \equiv \mathit{false}$.

   We want to show that Ps does not hold and $v \ulcorner s \urcorner \equiv \mathit{false}$:

   We assume $s \equiv I$ for the purpose of a contradiction. Then there are s' and t' such that $\ulcorner sI \urcorner = \ulcorner s't' \urcorner$ and $s' \equiv t'$, because one can choose s and I. Contradiction.

   Also: $v \ulcorner s \urcorner \equiv u\ (Q\ (\mathit{App} \ulcorner s \urcorner \ulcorner I \urcorner)) \equiv u \ulcorner\ulcorner s\ I \urcorner\urcorner \equiv \mathit{false}$. ■

4.4 Properties of *L*-Decidable Predicates

L-decidable predicates are closed under conjunction, disjunction and complement.

Lemma 4.12 *If P and Q are L-decidable, then $\boldsymbol{\lambda} s.\ Ps \wedge Qs$ is also L-decidable*

Proof Let u and v be deciders for P and Q respectively. Then $\lambda x.\mathit{andalso}\ (u\ x)\ (v\ x)$ is a decider for $\boldsymbol{\lambda} s.\ Ps \wedge Qs$, because if both $P\ s$ and $Q\ s$ hold for a term s, $u \ulcorner s \urcorner \equiv \mathit{true}$ as well as $v \ulcorner s \urcorner \equiv \mathit{true}$ and thus $(\lambda x.\mathit{andalso}\ (u\,x)\ (v\,x)) \ulcorner s \urcorner \equiv \mathit{andalso}\ (u \ulcorner s \urcorner)\ (v \ulcorner s \urcorner) \equiv \mathit{andalso\ true\ true} \equiv \mathit{true}$. In any other case $\mathit{andalso}\ (u \ulcorner s \urcorner)\ (v \ulcorner s \urcorner) \equiv \mathit{false}$ and $\neg(Ps \wedge Qs)$. ■

Lemma 4.13 *If P and Q are L-decidable, then $\boldsymbol{\lambda} s.\ Ps \vee Qs$ is also L-decidable.*

Proof Let u and v be deciders for P and Q respectively. Then $\lambda x.\mathit{orelse}\ (u\ x)\ (v\ x)$ is a decider for $\boldsymbol{\lambda} s.\ Ps \vee Qs$. ■

Lemma 4.14 *If P is L-decidable, then $\overline{P}$ is L-decidable.*

Proof Let u be a decider for P. Then $\lambda x.(u\ x)\ \mathit{false\ true}$ is a decider for $\overline{P}$: Assume for a term s that Ps holds and $u \ulcorner s \urcorner \equiv \mathit{true}$, thus $\overline{Ps}$ does not hold. Then $(\lambda x.(u\ x)\ \mathit{false\ true})\ u \ulcorner s \urcorner \equiv (u \ulcorner s \urcorner)\ \mathit{false\ true} \equiv \mathit{true\ false\ true} \equiv \mathit{false}$. The other case works parallel and for $\neg Ps$ (thus $\overline{P}s$) and $u \ulcorner s \urcorner \equiv \mathit{false}$ that $(\lambda x.(u\ x)\ \mathit{false\ true})\ u \ulcorner s \urcorner \equiv \mathit{true}$. ■

4.5 Coq-Decidability does not imply *L*-Decidability

It is common belief that the law of strong excluded middle is independent in constructive type theory. It states that every predicate is Coq-decidable. If Coq-Decidability implied *L*-Decidability, the consistent assumption of the law of strong excluded middle would yield that every predicate is *L*-decidable, which clearly is a contradiction to the fact that the self-halting problem is *L*-undecidable.

In Section 6.5 we will show that *L*-decidability implies Coq-decidability.

4.6 Discussion

The resulting theory of computation is based on *L* as programming language and Coq with its constructive type theory as the logical system. We call this theory **Constructive Computability Theory (CCT)**. Note that when we are talking of a predicate to be *L*-decidable this needs to be read as *the existence of a decider in L is provable in Coq*. By using Coq with excluded middle, strong excluded middle or another classical logic we would get a different theory where we could prove more predicates to be decidable or undecidable.

By basing our studies on a constructive theory we hope to get a more detailed insight on the different notions of computation, because missing assumptions always need to be made explicit.

Chapter 5

Acceptability and Rice's Theorem

5.1 Acceptable Predicates

We have defined the notion of L-decidability in Section 4.1. In this chapter we define a weaker notion of decidability, namely L-acceptability, which is often referred to as semi-decidability or recursive enumerability in the literature [11].

Definition 5.1 *We say that a term u* ***accepts a term*** *s and write $\pi\ u\ s$ if the application $u\ \ulcorner s \urcorner$ converges. We say that a term u* ***accepts a predicate*** *P if $P\ s\ \leftrightarrow\ \pi\ u\ s$. We call u the* ***acceptor*** *of P. A precicate is* ***L-acceptable*** *if it has an acceptor.*

If we want to say that $\overline{P}$ is L-acceptable, we may refer to P as ***L-coacceptable****. If we can prove that P has no acceptor, we say that P is* ***L-unacceptable****.*

Fact 5.2 *Let $s_1 \equiv s_2$. Then $\pi\ s_1\ t\ \leftrightarrow\ \pi\ s_2\ t$.*

Note that if a predicate P is L-decidable, both P and $\overline{P}$ are L-acceptable. One can turn the decision of the decider to a converging or diverging term.

Lemma 5.3 *If P is L-decidable, then both P and $\overline{P}$ are L-acceptable.*

Proof If u is a decider for P the procedure $v := \lambda x.u\ x\ I\ (\lambda\Omega)\ I$ is an acceptor for P and $v' := \lambda x.u\ x\ (\lambda\Omega)\ I\ I$ one for $\overline{P}$. ∎

This means that if a predicate is not L-acceptable, neither the predicate nor its complement can be L-decidable.

5.2 The Self-Halting Problem Reconsidered

We already know that the self-halting problem is L-undecidable (Theorem 4.3). We now show that it is also L-unacceptable.

Lemma 5.4 *$\boldsymbol{\lambda} t.\ t \ulcorner t \urcorner \Uparrow$ is not L-acceptable.*

Proof Assume an acceptor u, that is $\forall t.\ \ \pi\ u\ t\ \leftrightarrow\ t \ulcorner t \urcorner \Uparrow$.

If u is applied to its own encoding we have:

$$\pi\ u\ u\ \leftrightarrow\ u \ulcorner u \urcorner \Uparrow\ \leftrightarrow\ \neg \pi\ u\ u$$

A contradiction. Thus $\boldsymbol{\lambda} t.\ t \ulcorner t \urcorner \Uparrow$ is not L-acceptable. ■

The following lemma is shown analogous to Lemma 5.4.

Lemma 5.5 *$\boldsymbol{\lambda} t.\ \mathbf{C} t \wedge\ s \ulcorner s \urcorner \Uparrow$ is not L-acceptable.*

5.3 Properties of *L*-Acceptable Predicates

L-acceptable predicates are closed under conjunction and disjunction. We will now only prove the former and postpone the latter to Section 10.2.

Lemma 5.6 *If P and Q are L-acceptable predicates, then $\boldsymbol{\lambda} t.\ Pt \wedge Qt$ is L-acceptable.*

Proof Let u and v be acceptors for P and Q. Then $\lambda x.(\lambda y.u\,x)\ (v\,x)$ is an acceptor for $\boldsymbol{\lambda} t.\ P\ t \wedge Q\ t$, because $P\ t \wedge Q\ t \leftrightarrow \pi\ u\ t \wedge \pi\ v\ t$. Then there are procedures v_1 and v_2 such that $u \ulcorner t \urcorner \Downarrow v_1$ and $v \ulcorner t \urcorner \Downarrow v_2$, thus $(\lambda x.(\lambda y.u\,x)\ (v\,x)) \ulcorner t \urcorner \equiv (\lambda y.u \ulcorner t \urcorner)\ (v \ulcorner t \urcorner) \equiv (\lambda y.v_1) v_2 \equiv {v_1}^{y}_{v_2}$ which still is a procedure. ■

We will see that *L*-acceptable predicates are not closed under complement, since the self-halting predicate is *L*-acceptable, but not *L*-coacceptable.

5.4 Rice's Theorem

Scott's Theorem gives a criterion for the *L*-unacceptability of predicates satisfied only by combinators based on reduction equivalence. It nevertheless makes no strong statement over the behaviour of terms. We want to prove Rice's Theorem, that requires the predicate to be closed under acceptance instead of equivalence. It is due to H.G. Rice who proved it in his dissertation 1951 [17].

This has consequences in many fields, for instance in program verification, where its application yields the fact that there is no general method of proving a program property. The key difference between Scott's Theorem and Rice's Theorem is that Scott's Theorem talks about intensional program properties, such as structure of the program or convergence of the program itself, while Rice's Theorem makes a statement over extensional properties, such as the behaviour under an input.

To prove Rice's Theorem we need some preparation.We first need to show that closedness of a term is L-decidable. Then we need to internalize the encoding for natural numbers and the term encoding defined in the sections 3.2.1 and 3.2.2.

Lemma 5.7 *It is L-decidable if a term is k-closed.*

Proof One can easily construct a procedure that analyses all free variables of a term. ■

Corollary 5.8 *Closedness of a term is L-decidable.*

Lemma 5.9 *There are combinators P internalizing the function $n \mapsto \overline{n}$ and Q internalizing the function $s \mapsto \ulcorner s \urcorner$, that is:*

1. $P\,\overline{n} \equiv \ulcorner \overline{n} \urcorner$ *for all n.*
2. $Q \ulcorner s \urcorner \equiv \ulcorner \ulcorner s \urcorner \urcorner$ *for all s.*

Proof We define P and Q recursively such that the following equivalences hold:

$$\begin{aligned}
P\,\overline{0} &\equiv \ulcorner 0 \urcorner \\
P\,\overline{Sn} &\equiv \mathit{Lam}\,(\mathit{Lam}\,(\mathit{App} \ulcorner 0 \urcorner\,(P\,\overline{n}))) \\
Q \ulcorner n \urcorner &\equiv \mathit{Lam}\,(\mathit{Lam}\,(\mathit{Lam}\,(\mathit{App} \ulcorner 2 \urcorner\,(P\,\overline{n})))) \\
Q \ulcorner st \urcorner &\equiv \mathit{Lam}\,(\mathit{Lam}\,(\mathit{Lam}\,(\mathit{App}\,(\mathit{App} \ulcorner 1 \urcorner\,(Q \ulcorner s \urcorner))\,(Q \ulcorner t \urcorner)))) \\
Q \ulcorner \lambda s \urcorner &\equiv \mathit{Lam}\,(\mathit{Lam}\,(\mathit{Lam}\,(\mathit{App} \ulcorner 0 \urcorner\,(Q \ulcorner s \urcorner))))
\end{aligned}$$

■

We formulate an intuitionistic version of Rice's Theorem, which gives criteria for the L-unacceptability. The classical version then follows as a corollary.

Theorem 5.10 (Rice) *Let P be a predicate such that:*

1. *P is only satisfied by procedures: $P \subseteq \mathbf{P}$.*
2. *P is extensional: If s_1 and s_2 are procedures such that $\forall t. \pi\ s_1\ t \leftrightarrow \pi\ s_2\ t$, then $P\ s_1 \to P\ s_2$.*
3. *P is nontrivial: There are procedures s_1 and s_2 with $P\ s_1$ and $\neg(P\ s_2)$.*

Then we have the following:

1. *If $P(\lambda\Omega)$, then P is not L-acceptable*
2. *If $\neg P(\lambda\Omega)$, then $\overline{P}$ is not L-acceptable*

Proof Let t_1 and t_2 be procedures such that $P\ t_1$ and $\neg(P\ t_2)$.

1. Assume a procedure u that accepts P, so $\pi\ u\ t\ \leftrightarrow\ Pt$. We already know that we can have an acceptor c for the closedness-predicate by the lemmas 5.8 and 5.3, that means: $\pi\ c\ t\ \leftrightarrow\ t\ \mathit{closed}$.

   Let s be a term.

   Define
   $$v_s\ :=\ \lambda y.(\lambda\ t_2\ y)\ (s\ \ulcorner s \urcorner)$$

   We show $Pv_s\ \leftrightarrow\ \neg\pi ss$ as follows:

   (a) Assume Pv_s and πss. Then $\forall t.\pi\ v_s\ t\ \leftrightarrow\ \pi\ t_2\ t$. Thus Pt_2. Contradiction.

   (b) Let $\neg\pi\ s\ s$. Then $\forall t.\pi\ v_s\ t\ \leftrightarrow\ \bot\ \leftrightarrow\ \pi\ (\lambda\Omega)\ t$. Thus Pv_s since $P(\lambda\Omega)$.

   Define
   $$v := \lambda x.\ u\,(\mathit{Lam}\ (\mathit{App}\ \ulcorner \lambda t_2 1 \urcorner\ (\mathit{App}\ x\ (Qx))))$$
   $$v' := \lambda x.\mathit{andalso}\ (v\ x)\ (c\ x)$$

   Assume $\pi v' s$, which is the case if and only if $\pi\ v\ s \wedge \mathbf{C}s$. By the correctness of *Lam* and *App* this is equivalent to $\pi uv_s \wedge \mathbf{C}s$. But u accepts v_s if and only if Pv_s and we have already shown that $Pv_s\ \leftrightarrow\ \neg\pi ss$. Thus $\pi v' s\ \leftrightarrow\ \neg\pi\ s\ s \wedge \mathbf{C}s$.

   This contradicts that the self-halting problem for combinators is not *L*-acceptable (Lemma 5.5).

2. Let $\neg P(\lambda\Omega)$ Assume an acceptor u for $\overline{P}$.

   Let s be a term. Define
   $$v_s\ :=\ \lambda y.(\lambda\ t_1\ y)\ (s\ \ulcorner s \urcorner)$$

   We show $\neg Pv_s\ \leftrightarrow\ \neg\pi ss$ as follows:

   (a) Assume $\neg(Pv_s)$ and $\pi\ s\ s$. Then $\forall t.\pi\ v_s\ t\ \leftrightarrow\ \pi\ t_1\ t$. Thus Pv_s. Contradiction.

   (b) Let $\neg\pi\ s\ s$. Then $\forall t.\pi\ v_s\ t \leftrightarrow \bot \leftrightarrow \pi\ (\lambda\Omega)\ t$. Thus $\neg(Pv_s)$ since $\neg(P(\lambda\Omega))$.

   Define
   $$v := \lambda x.\ u\,(\mathit{Lam}\ (\mathit{App}\ \ulcorner \lambda t_1 1 \urcorner\ (\mathit{App}\ x\ (Qx))))$$
   $$v' := \lambda x.\mathit{andalso}\ (v\ x)\ (c\ x)$$

Similar to the first case we have

$$\begin{aligned} \pi\ v'\ s &\leftrightarrow \pi\ v\ s \land \mathbf{C}\ s \\ &\leftrightarrow \pi\ u\ v_s \land \mathbf{C}\ s \\ &\leftrightarrow P\ v_s \land \mathbf{C}\ s \\ &\leftrightarrow \neg\pi\ s\ s \land \mathbf{C}\ s \end{aligned}$$

This contradicts Lemma 5.5. ■

Corollary 5.11 (Rice, classical) *Let P be as in Theorem 5.10. Then P is L-undecidable.*

Proof Assume $P \subseteq \mathbf{P}$ is an extensional and nontrivial L-decidable predicate. Use the decider u to decide if P is satisfied by $\lambda\Omega$.

If this is the case, then we know that P is not L-acceptable by Theorem 5.10 (first case). Contradiction, because we assumed P were L-decidable.

If $\lambda\Omega$ does not satisfy P, we know that the complement of P is not L-acceptable, by Theorem 5.10 (second case). Contradiction.

Thus P is not L-decidable if it is a nontrivial extensional predicate. ■

We show several corollaries of Rice's Theorem:

Corollary 5.12 *$\boldsymbol{\lambda} s.\ \mathbf{P}s \land \forall t. \pi\ s\ t$ is not L-acceptable.*

Proof $\boldsymbol{\lambda} s.\ \mathbf{P}s \land \forall t. \pi\ s\ t$ is extensional and only satisfied by procedures. The term λI satisfies P, because it converges on all inputs. The term $\lambda\Omega$ does not satisfy P, because it diverges on every input. Thus $\boldsymbol{\lambda} s.\ \mathbf{P}s \land \forall t. \pi\ s\ t$ is not L-acceptable by Theorem 5.10. ■

Corollary 5.13 *$\boldsymbol{\lambda} s.\ \mathbf{P}s \land \exists t.\ \pi\ s\ t$ is not L-acceptable.*

Proof $\boldsymbol{\lambda} s.\ \mathbf{P}s \land \exists t.\ \pi\ s\ t$ is an extensional predicate only satisfied by procedures. It is satisfied by λI and not satisfied by $\lambda\Omega$. Thus by the first part of Theorem 5.10 it is not L-acceptable. ■

The following corollaries do all follow from Rice's Theorem with parallel reasoning:

Corollary 5.14 *$\boldsymbol{\lambda} s.\ \mathbf{P}s \land \exists t.\ \pi\ s\ t$ is L-undecidable.*

Corollary 5.15 *$\boldsymbol{\lambda} s.\ \mathbf{P}s \land \forall t.\ \neg\pi\ s\ t$ is not L-acceptable*

Corollary 5.16 *For every term t: $\boldsymbol{\lambda} s.\ := \mathbf{P}s \land \pi\ s\ t$ is L-undecidable.*

Corollary 5.17 *For every term t: $\boldsymbol{\lambda} s.\ := \mathbf{P}s \land \neg\pi\ s\ t$ is not L-acceptable.*

5.4.1 Rice's Theorem as a Corollary of Scott's Theorem

We can formulate the classical version of Rice's theorem in a slightly different way, so that it follows from Scott's theorem:

Corollary 5.18 *P is L-undecidable if it satisfies the following conditions:*

1. *P is only satisfied by combinators: $P \subseteq \mathbf{C}$.*
2. *P is extensional: If s_1 and s_2 are procedures such that $\forall t. \pi\ s_1\ t \leftrightarrow \pi\ s_2\ t$, then $P\ s_1 \rightarrow P\ s_2$.*
3. *P is nontrivial: There are procedures s_1 and s_2 with $P\ s_1$ and $\neg(P\ s_2)$.*

Proof We apply Scott's Theorem and need to show:

1. $P \subseteq \mathbf{C}$, which holds.
2. P is closed under reduction equivalence: P is even closed under acceptance, and thus under reduction equivalence.
3. There is a combinator satisfying P, which holds by the assumptions.
4. There is a combinator not satisfying P, which again holds by the assumptions.

■

It seems that our original version of Rice's theorem is not obtainable by Scott's theorem nor vice versa.

Chapter 6

Step-Indexed Evaluation

We have formally defined the semantics of L in Section 2.1 and proved several properties. Since the evaluation of programs in L may diverge it is not possible to write a function in Coq that evaluates programs directly. We thus use a technique called step indexing where a potentially non-total function gets equipped with an additional parameter that bounds the recursion depth.

We first give a definition for this step-indexed semantics as a predicate. Then we implement a step-indexed evaluation function and show that evaluation in all three semantics is equivalent.

6.1 The Step Evaluation Predicate

We already defined that $s \Downarrow t$ if $s \succ^* t$ and t is a procedure. We define $s \Downarrow^n t$ now, which means that $s \Downarrow t$, but with an evaluation path which is bounded in length dependent on n.

$$\frac{}{\lambda s \Downarrow^n \lambda s} \qquad \frac{s \Downarrow^n \lambda u \qquad t \Downarrow^n v \qquad u^0_v \Downarrow^n w}{st \Downarrow^{Sn} w}$$

For $s \Downarrow^n t$ we say that s evaluates in n steps to t. Note that this is not the same as $s \succ^n t$. We want to show that $s \Downarrow t \leftrightarrow \exists n. s \Downarrow^n t$.

We show the direction from left to right first, which needs two intermediate results:

Fact 6.1 *If $s \Downarrow^n t$, then $s \Downarrow^{Sn} t$.*

Lemma 6.2 *If $s \succ t$ and $t \Downarrow^n v$, then $s \Downarrow^{Sn} v$.*

Proof By an induction over the step-predicate. ∎

Lemma 6.3 *If $s \Downarrow t$, then there is n such that $s \Downarrow^n t$.*

Proof We know that $s \succ^* t$ and t is an abstraction. By an induction over $s \succ^* t$:

- If $s = t$, then s is obviously also an abstraction and thus for every n: $s \Downarrow^n s \leftrightarrow s \Downarrow^n t$.
- Assume $s \succ s' \succ^* t$ and there is n with $s' \Downarrow^n t$. Now by Lemma 6.2 we know that $s \Downarrow^{Sn} t$.

■

The converse is easier:

Lemma 6.4 *If $s \Downarrow^n t$ then $s \Downarrow t$.*

Proof By induction over the step-evaluation predicate:

- If $s = t$, then obviously s is also a procedure and $s \Downarrow s$ holds.
- Assume $s \succ^* \lambda u$, $t \succ^* v$ and $u_v^0 \succ^* w$. We need to show that $s\, t \succ^* w$, which is the case since $s\, t \succ^* (\lambda u)v \succ^* u_v^0 \succ^* w$. ■

Corollary 6.5 $s \Downarrow t \leftrightarrow \exists n. s \Downarrow^n t$

6.2 An Executable Semantics for *L*

After we have defined the step evaluation predicate we define a function *eva* now that is able to compute t given s and n such that $s \Downarrow^n t$. To simulate that *eva* is partial we let it compute something of type $\mathbf{T}_\perp$, which means that it either returns $\lfloor t \rfloor$ for some $t : \mathbf{T}$ or $\perp$ (to represent undefinedness).

The definition of *eva* is straightforward:

$$
\begin{aligned}
&\mathit{eva} : \mathbb{N} \to \mathbf{T} \to \mathbf{T}_\perp \\
&\mathit{eva}\ n\ k = \perp \\
&\mathit{eva}\ n\ (\lambda s) = \lfloor \lambda s \rfloor \\
&\mathit{eva}\ 0\ (st) = \perp \\
&\mathit{eva}\ (Sn)\ (st) = \mathsf{match}\ \mathit{eva}\ n\ s,\ \mathit{eva}\ n\ t\ \mathsf{with} \\
&\qquad \mid \lfloor \lambda s \rfloor, \lfloor t \rfloor \Rightarrow \mathit{eva}\ n\ s_t^0 \\
&\qquad \mid \_\ \_ \Rightarrow \perp
\end{aligned}
$$

We show that *eva* really computes the $\Downarrow^n$ predicate. Then we know that we are able to compute a normal form for a term, given it exists and we know how far to compute.

The correctness proof factors into several steps. We first show that any time *eva* evaluates to a term, this is really a procedure. Then we show that every value *eva* n s finds is indeed a term t such that $s \Downarrow^n t$. The converse statement is that if $s \Downarrow^n t$ also *eva* n $s = \lfloor t \rfloor$.

Lemma 6.6 *If eva n $s = \lfloor t \rfloor$ then t is a procedure.*

Proof By induction over n and case analysis over s and all recursive calls. ■

Lemma 6.7 *If eva n $s = \lfloor t \rfloor$ then $s \Downarrow^n t$.*

Proof Analogous to Lemma 6.6. ■

Lemma 6.8 *If $s \Downarrow^n t$ then eva n $s = \lfloor t \rfloor$.*

Proof By induction over $s \Downarrow^n t$. ■

Important is the following monotonicity property that shows that the value of *eva* does not rely on the index n:

Lemma 6.9 *eva* n $s = \lfloor t \rfloor \rightarrow$ *eva* (Sn) $s = \lfloor t \rfloor$.

Proof *eva* n $s = \lfloor t \rfloor$ implies $s \Downarrow^n t$, thus $s \Downarrow^{Sn} t$ holds and yields *eva* (Sn) $s = \lfloor t \rfloor$. ■

6.3 Internalized Maybe-Type

Our definition of *eva* returns a value of type $\mathbf{T}_\perp$. We thus need to internalize this type before we can internalize the *eva*-function. We can even generalize this to arbitrary types $X_\perp$.

We simply represent $\perp$ as $\lambda sn.n$ and $\lfloor t \rfloor$ as $\lambda sn.s \ulcorner t \urcorner$, where we can use any encoding instead of $\ulcorner . \urcorner$.

We define the two constructors:

$$\begin{aligned} \mathit{Some} &:= \lambda t\, s\, n.s\ t \\ \mathit{None} &:= \lambda s\, n.n \end{aligned}$$

The following two facts are important:

Fact 6.10 *For all procedures v: Some $v \not\equiv$ None.*

Fact 6.11 *For all procedures v and w: Some $v \equiv$ Some $w \rightarrow v = w$.*

6.4 Internalized Self Interpreter

Our goal is to internalize a self-interpreter for L. We first need to internalize the *eva* function. This will take several steps. We need to internalize every function that gets used in the *eva* function. This is the substitution function, that itself relies on a equality check on natural numbers. Thus we need to give combinators internalizing substitution and natural number equality.

Lemma 6.12 (Internalized Equality of Natural Numbers)
There is a combinator EqN such that for all natural numbers m and n one of the two following statements holds:

1. $m = n$ *and* $EqN\,\overline{m}\,\overline{n} \equiv true$
2. $m \neq n$ *and* $EqN\,\overline{m}\,\overline{n} \equiv false$

Proof Define *EqN* to satisfy the following equivalences:

$$\begin{array}{rclcrcl} EqN\,\overline{0}\;\overline{0} & \equiv & true & \qquad & EqN\,\overline{Sm}\;\overline{0} & \equiv & false \\ EqN\,\overline{0}\;\overline{Sn} & \equiv & false & & EqN\,\overline{Sm}\;\overline{Sn} & \equiv & EqN\,\overline{m}\;\overline{n} \end{array}$$

The proof is just induction over m. ■

Lemma 6.13 (Internalized Substitution)
There is a combinator Subst such that $Subst\,\ulcorner s \urcorner\,\overline{n}\,\ulcorner u \urcorner \equiv \ulcorner s^n_u \urcorner$.

Proof Define *Subst* recursively such that the following equivalences hold:

$$\begin{array}{rcl} Subst\,\ulcorner n \urcorner\,\overline{k}\,\ulcorner u \urcorner & \equiv & EqN\,\overline{n}\;\overline{k}\,\ulcorner u \urcorner\,(Var\,\overline{n}) \\ Subst\,\ulcorner st \urcorner\,\overline{k}\,\ulcorner u \urcorner & \equiv & App\,(Subst\,\ulcorner s \urcorner\,\overline{k}\,\ulcorner u \urcorner)\,(Subst\,\ulcorner t \urcorner\,\overline{k}\,\ulcorner u \urcorner) \\ Subst\,\ulcorner \lambda s \urcorner\,\overline{k}\,\ulcorner u \urcorner & \equiv & Lam\,(Subst\,\ulcorner s \urcorner\,(Succ\,\overline{k})\,\ulcorner u \urcorner) \end{array}$$

■

Theorem 6.14 (Internalized Step-Indexed Evaluation)
There is a combinator Eva such that $Eva\;\overline{k}\,\ulcorner s \urcorner \equiv \ulcorner eva\ k\ s \urcorner$.

Proof Define *Eva* recursively such that the following equivalences hold:

$$\begin{aligned}
Eva\ \overline{k}\ \ulcorner n \urcorner &\equiv \ulcorner \bot \urcorner \\
Eva\ \overline{k}\ \ulcorner \lambda s \urcorner &\equiv Some\ (Lam\ \ulcorner s \urcorner) \\
Eva\ \overline{0}\ \ulcorner st \urcorner &\equiv \ulcorner \bot \urcorner \\
Eva\ \overline{Sn}\ \ulcorner st \urcorner &\equiv Eva\ \overline{n}\ \ulcorner s \urcorner \\
&\quad (\lambda x.\ Eva\ \overline{n}\ \ulcorner t \urcorner \\
&\qquad (\lambda y.\ x\ (\lambda \ulcorner \bot \urcorner) \\
&\qquad\qquad (\lambda\lambda \ulcorner \bot \urcorner) \\
&\qquad\qquad (\lambda z.\ Subst\ z\ \overline{0}\ y))) \\
&\qquad \ulcorner \bot \urcorner \\
&\quad \ulcorner \bot \urcorner
\end{aligned}$$

■

6.5 L-Decidability Implies Coq-Decidability

We show that every *L*-decidable predicate is Coq-decidable. On the first view this is a surprising result, since *L* is a Turing-complete language while Coq is restricted to primitive recursive functions. *L*-decidability is that weak because it needs a decider in *L* and a proof of the correctness of the decider in Coq, as was already pointed out in chapter 4. One can now construct a deciding function in Coq based on this proof.

For the result we will need Constructive Choice, which can be found in the Coq standard library [5]. We will state it as a lemma but not prove it:

Lemma 6.15 (Constructive Choice) *Let $P : \mathbb{N} \to$ **Prop** be a computationally decidable predicate. Then we can build a function c of type $(\exists n. P\ n) \to$ $\mathbf{T}$ such that for every proof $H : \exists n. P\ n$ it holds that $P\ (cH)$.*

One can use this function c to get a witness for the satisfiability of a predicate by just proving the existence of one. Note that this is not trivial in constructive mathematics, since the witness really needs to be computed.

Theorem 6.16 *If P is an L-decidable predicate, it is Coq-decidable.*

Proof Let P be an *L*-decidable predicate, that means there is $u : \mathbf{T}$ with $\forall s. u \ulcorner s \urcorner \equiv$ *true* $\wedge\ P\ s \vee u \ulcorner s \urcorner \equiv$ *false* $\wedge \neg(P\ s)$. We want to prove that for an arbitrary $s : \mathbf{T}$ we either can compute a proof for $P\ s$ or for $\neg(P\ s)$.

It is easy to see that the following holds

$$\exists n. u \ulcorner s \urcorner \Downarrow^n true\ \vee\ u \ulcorner s \urcorner \Downarrow^n false$$

Then we can use the constructive choice function c to compute such an n.

We can simply consider the term t with $u\ulcorner s\urcorner \Downarrow^n t$. If $t =$ *true*, we can show that Ps holds: We either have $u\ulcorner s\urcorner \equiv$ *true* $\wedge\ Ps$, which proofs what we want. Or we have $u\ulcorner s\urcorner \equiv$ false, but then *true* $\equiv$ $false$ which would mean that *true* $=$ *false*. Contradiction.

If $t \neq$ *true*, we have $t = false$ and thus $\neg(P\ s)$ holds by a similar reasoning. ∎

Chapter 7

Self-Interpretation

Our goal of an internalized self-interpreter for L is almost reached. We defined a step-indexed evaluation predicate, which is implementable in Coq. The next natural step is to internalize this function.

Then the question rises how to turn step-indexed functions into combinators that ignore the bound. In general this can be solved using a combinator that turns total step-indexed combinators into partial ones.

7.1 On the Internalization of Partial Functions

We come back to the goal of internalizing a function. Remember that a combinator u internalizes a function $f : \mathbf{T} \to \mathbf{T}$ if for every term s: $u \ulcorner s \urcorner \equiv \ulcorner f\ s \urcorner$.

We have already seen how to internalize total Coq-functions in Section 3.3. The next question is how to generalize this to arbitrary functions.

There are two ways to solve the problem for partial functions. One way is to internalize the function directly by translating the recursive specification carefully into L, in parallel to the internalization in Section 3.3. We will use this method in Section 8.1.

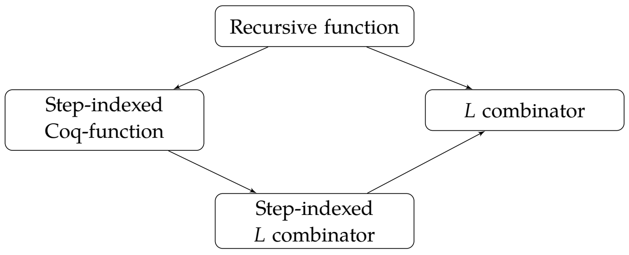
The alternative way involves more intermediate steps, but is easier to verify. One first implements the function f in Coq using step-indexing. This results in a total function $f' : \mathbb{N} \to \mathbf{T} \to \mathbf{T}_\perp$, that can be internalized to a procedure F using the known methods. The term F then has the following properties:

1. F is total in the sense that $\forall nt. F\ \overline{n} \ulcorner t \urcorner \equiv \mathit{None} \lor \exists v. \mathbf{P} v \land F\ \overline{n} \ulcorner t \urcorner \equiv \mathit{Some} \ulcorner v \urcorner$.
2. F is monotone in the sense that $\forall ntv. F\ \overline{n} \ulcorner t \urcorner \equiv \mathit{Some}\ v \to F\ \overline{Sn} \ulcorner t \urcorner \equiv \mathit{Some}\ v$.

The first property is immediately clear, because F internalized the total function f'. The second property follows from the fact, that f' would behave like f if one would simply ignore the index. The result of F thus does not depend on the index n.

The basic idea of this approach is a simple linear search over the index n of F.

The two internalization strategies are visualized in the following diagram. Note that the diagram is commuting in a semantic sense, as that the resulting combinators are semantically equivalent. They are, however, not in general equivalent in terms of L. Both strategies have advantages and disadvantages. One especially needs to consider if the fiddlier proofs in the direct version outweigh the work to build a step-indexed version. The step-indexed version just works with functions f of type $\mathbf{T} \to \mathbf{T}$ and needs to be adapted for other types.



Internalization strategies

We define a combinator, that takes a monotone and total procedure F and an input t and tries out indices n until it finds one such that $F\ \overline{n}\ \ulcorner t \urcorner \equiv \mathit{Some}\ v$. Then we have found the value $ft = v$. This is done using a simple linear search:

We define S' such that:

$$S'\ \overline{n}\ F\ \ulcorner t \urcorner \equiv (F\ \overline{n}\ \ulcorner t \urcorner)\ K\ ((\lambda x.(\lambda x.S'\ x)\ (\mathit{Succ}\ \overline{n})\ F\ \ulcorner t \urcorner\ x))$$

We assume that F is total and monotone from now on:
Lemma 7.1

$$S'\ \overline{n}\ F\ \ulcorner t \urcorner \equiv S'\ \overline{Sn}\ F\ \ulcorner t \urcorner$$

Proof By a case analysis of $F\ \overline{n}\ \ulcorner t \urcorner$.

If it is $\mathit{Some}\ v$, we know because of the monotonicity that $F\ \overline{Sn}\ \ulcorner t \urcorner \equiv \mathit{Some}\ v$ as well. But then $S'\ \overline{n}\ F\ \ulcorner t \urcorner \equiv v \equiv S'\ \overline{n}\ F\ \ulcorner t \urcorner$.

If $F\ \overline{n}\ \ulcorner t \urcorner \equiv \mathit{None}$ we directly have $S'\ \overline{n}\ F\ \ulcorner t \urcorner \equiv S'\ \overline{Sn}\ F\ \ulcorner t \urcorner$. ∎

We define the real combinator S as

$$S := S'\ \overline{0}$$

Lemma 7.2 *$S\ F\ \ulcorner t \urcorner \equiv S'\ \overline{n}\ F\ \ulcorner t \urcorner$ for any natural number n.*

Proof By induction over n and Lemma 7.1 ∎

The correctness of S factors into two parts. First we show that any time S finds a value v there really is an n such that $F\ \overline{n}\ \ulcorner t \urcorner \equiv \mathit{Some}\ v$ and vice versa. Secondly any time $S\ s\ \ulcorner t \urcorner$ converges, the resulting value is in fact closed. This yields a combinator that lets a step-indexed combinator forget about its bound. We list all necessary assumptions for the theorem:

Theorem 7.3 *If F is a procedure such that*

1. $\forall n\, t. F\ \overline{n}\ \ulcorner t \urcorner \equiv \mathit{None} \lor \exists v. \mathbf{P} v \land F\ \overline{n}\ \ulcorner t \urcorner \equiv \mathit{Some}\ \ulcorner v \urcorner$
2. $\forall n\, t\, v. F\ \overline{n}\ \ulcorner t \urcorner \equiv \mathit{Some}\ v \rightarrow\ F\ \overline{Sn}\ \ulcorner t \urcorner \equiv \mathit{Some}\ v$

we have for every procedure v and every term t:

$$S\ F\ \ulcorner t \urcorner \equiv v\ \leftrightarrow\ \exists n. F\ \overline{n}\ \ulcorner t \urcorner \equiv \mathit{Some}\ v$$

Proof The direction from right to left is trivial using Lemma 7.2.

The direction from left to right takes considerable effort and a new proof technique. Instead of the original statement we prove a generalized version:

$$\forall n. S'\ \overline{n}\ F\ \ulcorner t \urcorner \succ^k v \rightarrow\ \exists n. F\ \overline{n}\ \ulcorner t \urcorner \equiv \mathit{Some}\ v$$

Because $S\ F\ \ulcorner t \urcorner \equiv v$ where v is a procedure we know that there is a k such that $S\ F\ \ulcorner t \urcorner \succ^k v$.

We use strong induction over k, i.e. we assume that the following statement holds for all k' smaller than k:

$$\forall n. S'\ \overline{n}\ F\ \ulcorner t \urcorner \succ^{k'} v \rightarrow\ \exists n. F\ \overline{n}\ \ulcorner t \urcorner \equiv \mathit{Some}\ v$$

We then analyze if $F\ \overline{n}\ \ulcorner t \urcorner$ is equivalent to *Some* v' or to *None*. In both cases we assume that $S'\ \overline{n}\ s\ \ulcorner t \urcorner \succ^k v$ and show that there is a n such that $F\ \overline{n}\ \ulcorner t \urcorner \equiv \mathit{Some}\ v$.

Assume the first holds. Then we know that $S'\ \overline{n}\ \ulcorner t \urcorner \equiv v'$, but we also know that $S'\ \overline{n} \ulcorner t \urcorner \equiv v$. Both v and v' are abstractions and thus $v = v'$. Thus we have $F\ \overline{n} \ulcorner t \urcorner \equiv \mathit{Some}\ v$.

In the other case we know that $F\ \overline{n}\ \ulcorner t \urcorner \equiv \mathit{None}$. We thus know that $S'\ \overline{n}\ s\ \ulcorner t \urcorner \succ^m S'\ \overline{n}\ s\ \ulcorner t \urcorner$ for some $m > 0$.

From $S'\ \overline{n}\ s\ \ulcorner t \urcorner \succ^k v$ and $S'\ \overline{n}\ s\ \ulcorner t \urcorner \succ^m S'\ \overline{Sn}\ s\ \ulcorner t \urcorner$ we can conclude that there is $m' < k$ with $S'\ \overline{Sn}\ s\ \ulcorner t \urcorner \succ^{m'} v$. This gives us the ability to apply the inductive hypothesis and we are done. ∎

Note that the last proof would be a lot harder, if not impossible, if we did not have the parametric diamond property resulting from the uniform confluence of L.

The following lemma is needed and does not follow from Lemma 7.3. The theorem assumes the term v to be a procedure and this assumption can not be weakened. We are able to weaken it after we have shown the following independent result:

Lemma 7.4 *Let F be a procedure as in the theorem above. Then for every abstraction v we have that $S\ F \ulcorner t \urcorner \equiv v \to \mathbf{C}v$.*

Proof The proof is similar to the one above. We again generalize and assume $S'\ \overline{n}\ F \ulcorner t \urcorner \succ^k v$.

Either we have $F\ \overline{n} \ulcorner t \urcorner \equiv \mathit{Some}\ v'$ where v' is a procedure. Then we again conclude $v = v'$ and by this v is closed for sure.

In the other case we have $F\ \overline{n} \ulcorner t \urcorner \equiv \mathit{None}$, thus $S'\ \overline{n}\ s \ulcorner t \urcorner \succ^m S'\ \overline{n}\ s \ulcorner t \urcorner$ for some $m > 0$, which yields a $m' < k$ such that $S'\ \overline{Sn}\ s \ulcorner t \urcorner \succ^{m'} v$. By the inductive hypothesis we are done. ∎

This general technique of size induction over the length of the reduction path can always be used to verify partial combinators. If the combinator is defined using S there should be no need of such proofs, since everything one needs to do is create a monotone and total step-indexed combinator F and use the correctness theorems above.

Corollary 7.5 (Correctness of S) *For a procedure F as above and an abstraction v it holds that: $S\ F \ulcorner t \urcorner \Downarrow v \to \exists n. F\ \overline{n} \ulcorner t \urcorner \equiv \mathit{Some}\ v$*

Proof We need the fact that v is closed in both directions to apply Theorem 7.5. This follows from Lemma 7.4 for the direction from left to right and from Fact 6.11 in the other direction. ∎

Corollary 7.5 can be used to verify any combinator that has been defined via the combinator S. We will do this for the self-interpreter combinator.

7.2 Definition of the Self-Interpreter

An interpreter for L would be a partial function $I : \mathbf{T} \to \mathbf{T}$ such that $s \Downarrow Is$ if Is is defined and $s \Uparrow$ otherwise. Clearly, such an interpreter is not definable in Coq. But we can define a term *Eval* with $\mathit{Eval} \ulcorner s \urcorner \equiv \ulcorner t \urcorner \leftrightarrow s \Downarrow t$.

This is routine using the combinator S. We have already done the work of writing a step-indexed interpreter combinator in Section 6.4.

To use the theorems over S we need to verify that *Eva* is total and monotone:

Lemma 7.6 *Either* $Eva\ \overline{n}\ \ulcorner t \urcorner \equiv None$ *or there is a procedure* v *such that* $Eva\ \overline{n}\ \ulcorner t \urcorner \equiv Some\ v$.

Proof This is trivial, since *Eva* internalizes *eva*. ∎

Lemma 7.7 *For every procedure* v: $Eva\ \overline{n}\ \ulcorner t \urcorner \equiv Some\ v \to\ Eva\ \overline{Sn}\ \ulcorner t \urcorner \equiv Some\ v$.

Proof Follows with Lemma 6.9 and the correctness of *Eva*. ∎

We can define:

$$Eval := S(\lambda x.Eva\ x)$$

The η-expansion is not necessary, but convenient since it simplifies the formal proofs. We are interested in the the following two statements:

1. $Eval\ \ulcorner s \urcorner \equiv \ulcorner t \urcorner\ \leftrightarrow\ s \Downarrow t$
2. $Eval\ \ulcorner s \urcorner \Downarrow\ \leftrightarrow s \Downarrow$

Both statements are provable almost directly using the correctness of S. We first show an intermediate lemma:

Lemma 7.8 *For any procedure* v: $Eval\ \ulcorner s \urcorner \equiv v\ \leftrightarrow\ \exists n.Eva\ \overline{n}\ \ulcorner s \urcorner \equiv Some\ v$

Proof We use Corollary 7.5. We then have to show:

- $\exists n.(\lambda x.Eva\ x)\ \overline{n}\ \ulcorner s \urcorner \equiv Some\ v\ \leftrightarrow\ \exists n.Eva\ \overline{n}\ \ulcorner s \urcorner \equiv Some\ v$: This is trivial.
- $(\lambda x.Eva\ x)$ is monotone and total. This follows directly from the totality and monotonicity of *Eva*. ∎

We can show the two correctness statements of *Eval*:

Theorem 7.9 $Eval\ \ulcorner s \urcorner \equiv \ulcorner t \urcorner\ \leftrightarrow\ s \Downarrow t$

Proof For the direction from left to right assume that $Eva\ \ulcorner s \urcorner \equiv \ulcorner t \urcorner$ which is by Lemma 7.8 equivalent to $Eva\ \overline{n}\ \ulcorner s \urcorner \equiv Some\ \ulcorner t \urcorner$ for some $n : \mathbb{N}$. Now analyze $eva\ n\ s$. If it is $\lfloor t' \rfloor$ we can conclude that $\ulcorner \lfloor t' \rfloor \urcorner \equiv Some\ \ulcorner t \urcorner$ and thus $t' = t$ and $s \Downarrow t' = t$.

For the direction from right to left assume that $s \Downarrow t$, thus there is some n with $Eva\ \overline{n}\ \ulcorner s \urcorner \equiv \ulcorner t \urcorner$ by the correctness of *Eva*. This directly yields $Eval\ \ulcorner s \urcorner \equiv \ulcorner t \urcorner$ by Lemma 7.8. ∎

Theorem 7.10 $Eval\ \ulcorner s \urcorner \Downarrow\ \leftrightarrow s \Downarrow$

Proof We first show the easy direction from right to left. Assume s converges. Then there is n such that *eva* $n\ s = \lfloor t \rfloor$. This yields the convergence of *Eval* $\ulcorner s \urcorner$ since *Eval* $\ulcorner s \urcorner \equiv \ulcorner t \urcorner$ then.

For the direction from left to right assume that *Eval* $\ulcorner s \urcorner \Downarrow$, that is there is an abstraction v such that *Eval* $\ulcorner s \urcorner \equiv v$. We can use Lemma 7.8, so that we need to conclude $s \Downarrow$ from *Eva* $\overline{n}\ \ulcorner s \urcorner \equiv v$.

We also need to show that v is closed, because this was necessary for the usage of Lemma 7.8. We can use Lemma 7.4, which just requires the already proven monotonicity and totality of *Eva*. ∎

Chapter 8

Parallel Or and AD Theorem

It is a well known result of computability theory that a problem is decidable if and only if it is both acceptable and co-acceptable, see for instance [11]. We need to reformulate this statement to make it provable in our constructive setting. Thus we show that for a propositionally decidable predicate P we can conclude *L*-decidability from *L*-acceptability and *L*-coacceptability.

We will refer to this theorem as *AD Theorem*. The proof assumes acceptors u and v for P and $\overline{P}$ respectively, which are executed on a given input. The idea is a combinator *Por* that is able to simulate a parallel evaluation of two terms. Depending on which of the two terms converged, the value of the program is *true* or *false*. *Por* can be used to run u and v on an input and thus to construct a decider.

8.1 Specification of Parallel Or

We are interested in a combinator *Por* such that *Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner$ converges if and only if either s or t converges. Moreover, we want *Por* to signal which one of its inputs converged. That means if *Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner$ evaluates to *true*, then s converged, and if *Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner$ evaluates to *false*, t converged.

The informal idea of the algorithm is as follows: We evaluate s with an evaluation bound of 1 using our internalized step-indexed interpreter. If s converged, we are done. If not, we execute t with bound 1. If this again produced no result, we increase the bound by 1 and proceed.

This is in fact a linear search over all natural numbers, that evaluates both s and t for n steps one after another. This technique of interleaving the execution of two algorithms is known as dovetailing.

We already have everything we need for the definition of *Por* at hand. We just need to be careful on how to formulate the correctness criterion of *Por*.

We formulate it as:

For all terms s and t one of the following statements holds:

1. *If s converges or t converges, then Por $\ulcorner s \urcorner \ulcorner t \urcorner$ converges.*
2. *If Por $\ulcorner s \urcorner \ulcorner t \urcorner$ converges, then either Por $\ulcorner s \urcorner \ulcorner t \urcorner \equiv$ true and s converges or Por $\ulcorner s \urcorner \ulcorner t \urcorner \equiv$ false and t converges.*

We prove that such a combinator *Por* exists in 8.2. Note that the first part can not be strengthened in a way like *If s converges, then Por $\ulcorner s \urcorner \ulcorner t \urcorner \equiv$ true*, since this is not necessarily the case. Consider for instance $s := I\ I$ and $t := I$. Since t already is a procedure, it converges faster than s and thus *Por* $\ulcorner s \urcorner \ulcorner t \urcorner \equiv$ *false*.

First we define *Por*. We have already described the informal idea. We implement a combinator *Por′* that takes an encoded natural number $\overline{n}$ and two terms as input and executes the terms both to depth $\overline{n}$. If none of them converges, it continues its search with $\overline{Sn}$ as a new argument recursively. This idea is parallel to the one in Section 7.1, but instead of executing one combinator via the step-indexed self-interpreter and a linear search two combinators get executed.

We can then simply define *Por* to be *Por′* $\overline{0}$. In fact, we could even choose any index here.

Fact 8.1 *There is a combinator Por′ such that:*

$$\begin{aligned} Por'\ \overline{n}\ \ulcorner s \urcorner \ulcorner t \urcorner \ \equiv\ & Eva\ \overline{n}\ \ulcorner s \urcorner (\lambda\lambda\ true) \\ & (Eva\ \overline{n}\ \ulcorner t \urcorner (\lambda\lambda false) \\ & (\lambda(\lambda x.Por'\ x)\ (Succ\ \overline{n})\ \ulcorner s \urcorner \ulcorner t \urcorner)\ I \end{aligned}$$

We show some essential properties of *Por′* that lead to the correctness of *Por*.

8.2 Correctness and Completeness of Parallel Or

The index $\overline{n}$ for *Por′* is not interesting regarding convergence. This is expressed in the following lemma:

Lemma 8.2 *Por′ $\overline{n}\ \ulcorner s \urcorner \ulcorner t \urcorner$ converges if and only if Por′ $\overline{Sn}\ \ulcorner s \urcorner \ulcorner t \urcorner$ does.*

Proof If *eva* $n\ s$ is $\lfloor v \rfloor$, then *eva* $(Sn)\ s$ is also $\lfloor v \rfloor$ and both *Por′* $\overline{n}\ \ulcorner s \urcorner \ulcorner t \urcorner$ and *Por′* $\overline{Sn}\ \ulcorner s \urcorner \ulcorner t \urcorner$ evaluate to *true*.

Assume *eva* $n\ s$ is $\bot$. If *eva* $n\ t$ evaluates to a value, then *eva* $(Sn)\ t$ and both instances of *Por′* evaluate to *false*. In the case where both *eva* $n\ s$ and *eva* $n\ t$ are $\bot$, we even have *Por′* $\overline{n}\ \ulcorner s \urcorner \ulcorner t \urcorner \equiv$ *Por′* $\overline{Sn}\ \ulcorner s \urcorner \ulcorner t \urcorner$. ■

Using this we can conclude:

Corollary 8.3 *Por'* $\overline{0}\,\ulcorner s\urcorner\,\ulcorner t\urcorner$ *converges if and only if Por'* $\overline{n}\,\ulcorner s\urcorner\,\ulcorner t\urcorner$ *converges.*

It seems tempting to formulate Lemma 8.2 stronger with a statement like *Por'* $\overline{n}\,\ulcorner s\urcorner\,\ulcorner t\urcorner \equiv$ *Por'* $\overline{Sn}\,\ulcorner s\urcorner\,\ulcorner t\urcorner$. While this is unnecessary, it also does not hold. It may be the case, that there is no v such that $s \Downarrow^n v$, but there is one with $s \Downarrow^{Sn} v$. Then the right hand side of the proposed equivalence will evaluate to *true* in any case, while the left hand side evaluates to *false* if t converges in n steps.

Having this it is clear that if one of the terms converges, the parallel or of them converges. We show one version of this, while the second is exactly parallel:

Lemma 8.4 *If s converges, then Por* $\ulcorner s\urcorner\,\ulcorner t\urcorner$ *converges.*

Proof Assume that s converges, that is it evaluates in n steps to v. To show that *Por* $\ulcorner s\urcorner\,\ulcorner t\urcorner$ converges it is enough to show that *Por'* $\overline{n}\,\ulcorner s\urcorner\,\ulcorner t\urcorner$ converges. This is the case, because *Eva* $\overline{n}\,\ulcorner s\urcorner \equiv \ulcorner v\urcorner$ and thus *Por'* $\overline{n}\,\ulcorner s\urcorner\,\ulcorner t\urcorner \equiv$ *true*. ∎

Lemma 8.5 *If t converges, then Por* $\ulcorner s\urcorner\,\ulcorner t\urcorner$ *converges.*

Corollary 8.6 *If s or t converge, then Por* $\ulcorner s\urcorner\,\ulcorner t\urcorner$ *converges.*

This is the first direction of the correctness. The second works by induction over the reduction length. The proof is parallel to the correctness proof of the partial evaluation combinator S in Section 7.1. Note that while a definition of *Por* via S is possible, it is more complicated than our direct approach used here.

Lemma 8.7 *If Por* $\ulcorner s\urcorner\,\ulcorner t\urcorner$ *converges, then either Por* $\ulcorner s\urcorner\,\ulcorner t\urcorner \equiv$ *true and s converges or Por* $\ulcorner s\urcorner\,\ulcorner t\urcorner \equiv$ *false and t converges.*

Proof We strengthen the claim to

$$\forall n.\mathit{Por'}\ \overline{n}\,\ulcorner s\urcorner\,\ulcorner t\urcorner \succ^m v \to\ \mathit{Por'}\ \overline{n}\,\ulcorner s\urcorner\,\ulcorner t\urcorner \equiv \mathit{true} \land s \Downarrow \lor \mathit{Por'}\ \overline{n}\,\ulcorner s\urcorner\,\ulcorner t\urcorner \equiv \mathit{false} \land t \Downarrow$$

for an arbitrary procedure v.

The proof then uses strong induction over m. We can distinct four cases:

1. $s \Downarrow^n v$ and $t \Downarrow^n v'$. Then s converges and *Por'* $\overline{n}\,\ulcorner s\urcorner\,\ulcorner t\urcorner \equiv$ *true*, because *Eva* $\overline{n}\,\ulcorner s\urcorner \equiv \ulcorner v\urcorner$.
2. $s \Downarrow^n v$ and there is no v' with $t \Downarrow^n v'$. Then s converges and *Por'* $\overline{n}\,\ulcorner s\urcorner\,\ulcorner t\urcorner \equiv$ *true*, because *Eva* $\overline{n}\,\ulcorner s\urcorner \equiv \ulcorner v\urcorner$.
3. There is no v with $s \Downarrow^n v$ and we have v' with $t \Downarrow^n v'$. Then t converges and *Por'* $\overline{n}\,\ulcorner s\urcorner\,\ulcorner t\urcorner \equiv$ *false*, because *Eva* $\overline{n}\,\ulcorner t\urcorner \equiv \ulcorner v'\urcorner$.

4. There are neither v nor v' with $s \Downarrow^n v$ or $t \Downarrow^n v'$. Then $\mathit{Por}'\ \overline{n}\ \ulcorner s \urcorner\ \ulcorner t \urcorner \equiv \mathit{Por}'\ \overline{Sn}\ \ulcorner s \urcorner\ \ulcorner t \urcorner$ and there is $l < m$ such that $\mathit{Por}'\ \overline{n}\ \ulcorner s \urcorner\ \ulcorner t \urcorner \succ^l \mathit{Por}'\ \overline{Sn}\ \ulcorner s \urcorner\ \ulcorner t \urcorner$.

   We have to show that $(\mathit{Por}'\ \overline{n}\ \ulcorner s \urcorner\ \ulcorner t \urcorner \equiv \mathit{true} \wedge s \Downarrow) \vee (\mathit{Por}'\ \overline{n}\ \ulcorner s \urcorner\ \ulcorner t \urcorner \equiv \mathit{false} \wedge t \Downarrow)$, which is equivalent to $(\mathit{Por}'\ \overline{Sn}\ \ulcorner s \urcorner\ \ulcorner t \urcorner \equiv \mathit{true} \wedge s \Downarrow) \vee (\mathit{Por}'\ \overline{Sn}\ \ulcorner s \urcorner\ \ulcorner t \urcorner \equiv \mathit{false} \wedge t \Downarrow)$.

   We can invoke the inductive hypothesis here and are done. ■

Corollary 8.8 *If Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner \equiv$ *true then* s *converges.*

Proof Obviously *Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner$ converges, and thus we have that either *Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner \equiv$ *true* and s converges or *Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner \equiv$ *false* and t converges.

In the first case, we are done. In the second case we have a contradiction, since then *Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner$ is equivalent to *true* and *false*, but we know that *true* $\not\equiv$ *false*. ■

Corollary 8.9 *If Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner \equiv$ *false then* t *converges.*

Corollary 8.10 *Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner$ *converges if and only if either* s *or* t *converge.*

Proof The direction from left to right follows from Lemma 8.7. The direction from right to left is a direct consequence from the lemmas 8.4 and 8.5. ■

Theorem 8.11 (Existence of Parallel Or) *There is a combinator Por such that for all terms* s *and* t *one of the following statements holds:*

1. *If* s *converges or* t *converges, then Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner$ *converges.*
2. *If Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner$ *converges, then either Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner \equiv$ *true and* s *converges or Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner \equiv$ *false and* t *converges.*

Proof Follows directly from the last lemmas. ■

Remark

Classically, one would formulate the existence of a combinator like *Por* in a stronger way:

There is a combinator Por such that for all terms s *and* t *one of the following statements holds:*

1. s *converges and Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner \equiv$ *true.*
2. t *converges and Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner \equiv$ *false.*
3. *The terms* s, t, *and Por* $\ulcorner s \urcorner\ \ulcorner t \urcorner$ *all diverge.*

This is not possible in a constructive setting. It is obvious that the halting problem is not propositionally decidable in Coq. The assumption of a combinator with the properties stated above would now yield exactly that the halting predicate is propositionally decidable. To see this, just set s and t to the same term. Then examine which of the cases holds to find out if the term converged.

8.3 The AD-Theorem

Theorem 8.12 (AD) *If a propositionally decidable predicate is both L-acceptable and L-coacceptable, it is L-decidable.*

Proof Let P be an arbitrary predicate that is propositionally decidable, that is $\forall s : \mathbf{T}. Ps \lor \neg Ps$. Assume acceptors u and v for P and $\overline{P}$ respectively:

$$(Ps \leftrightarrow \pi\; u\; s) \land (\neg Ps \leftrightarrow \pi\; v\; s)$$

We define a decider for P as follows:

$$\lambda s.\mathit{Por}\;(\mathit{App}\,\ulcorner u \urcorner\;(Q\;s))\;(\mathit{App}\,\ulcorner v \urcorner\;(Q\;s))$$

Note that $(\lambda s.\mathit{Por}\;(\mathit{App}\,\ulcorner u \urcorner\;(Q\;s))\;(\mathit{App}\,\ulcorner v \urcorner\;(Q\;s)))\,\ulcorner s \urcorner \equiv \mathit{Por}\,\ulcorner u\,\ulcorner s \urcorner \urcorner\,\ulcorner v\,\ulcorner s \urcorner \urcorner$.

Since P is propositionally decidable, we either have $u\,\ulcorner s \urcorner$ converges or $v\,\ulcorner s \urcorner$ converges and thus $\mathit{Por}\,\ulcorner u\,\ulcorner s \urcorner \urcorner\,\ulcorner v\,\ulcorner s \urcorner \urcorner$ converges in both cases.

Thus by Lemma 8.7 either $(\lambda s.\mathit{Por}\;(\mathit{App}\,\ulcorner u \urcorner\;(Q\;s))\;(\mathit{App}\,\ulcorner v \urcorner\;(Q\;s)))\,\ulcorner s \urcorner \equiv \mathit{true}$ and Ps or $(\lambda s.\mathit{Por}\;(\mathit{App}\,\ulcorner u \urcorner\;(Q\;s))\;(\mathit{App}\,\ulcorner v \urcorner\;(Q\;s)))\,\ulcorner s \urcorner \equiv \mathit{false}$ and $\neg Ps$, which shows the proposition. ■

Chapter 9

Recursive Enumerability

An *L*-acceptable predicate is often called a *recursively enumerable (r.e.)* predicate in the literature. One usually shows that an acceptable problem can be seen as the codomain of a partial function and vice versa. We will define the notion of enumerability for *L* and show that it coincides with L-acceptability.

9.1 Enumerable Predicates

Definition 9.1 *A predicate $P : \mathbf{T} \to$ **Prop** is called **L-enumerable** if there is a combinator F with:*

1. $\forall n. F\,\overline{n} \equiv \mathit{None} \vee \exists s. F\,\overline{n} \equiv \mathit{Some}\,\ulcorner s \urcorner \wedge Ps$
2. $\forall s. Ps \to \exists n. F\,\overline{n} \equiv \mathit{Some}\,\ulcorner s \urcorner$

*We call F the **enumerator** of P.*

We show that *L*-enumerable and *L*-acceptable predicates are indeed exactly the same. The proof consists of two constructions.

We need a way to construct an acceptor out of an enumerator. We can use a method here which we have already seen and do linear search over the codomain of the enumerator.

The other way is harder and needs considerably more effort. We need to construct an enumerator out of an acceptor. To do so we first enumerate all terms and use an internalized bijection of type $\mathbb{N} \to \mathbf{T}$. We then show that there is a surjection $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$. Using this constructions the enumerator can work as follows: Under input n, it deconstructs the input to a pair of natural numbers m_1 and m_2. m_2 is used to construct a term t and the enumerator returns the result of $\mathit{Eva}\,\overline{m_1}\,\ulcorner u \ulcorner t \urcorner \urcorner$. Since the functions ($\mathbb{N} \to \mathbf{T}$ and $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$) are both bijective, every term gets executed with every evaluation depth eventually.

In the classical approach to computability theory it is essential that all terms can be enumerated, which is considered to be something trivial in general but fiddly if applied to special cases. In our constructive approach it is not enough to point to the obvious countability of terms to get a numbering, we need to explicitly construct a function $g : \mathbf{T} \to \mathbb{N}$ and its inverse $g^{-1} : \mathbb{N} \to \mathbf{T}$.

We could have used the well known ideas exploiting the properties of prime factorization used for gödelization, but this is also a good place to show the computational power of *L* by implementing lists and some well-known list functions. We possibly spend a little more effort for this than the usual approach would take, but additionally get an independent development of Scott-encoded lists in *L*.

This chapter thus splits into five sections. The first implements the functions g and g^{-1} in Coq and proves their correctness. The second one implements a bunch of list functions like *append*, *map*, *filter*, *nth* and *pos* in *L* before all this gets combined in section 3 to internalized combinators. The fourth section shows the equivalence of *L*-enumerability and *L*-Acceptability. We show that existential quantification over an *L*-decidable predicate is *L*-acceptable in Section 5.

9.2 Enumeration of terms

We are interested in a bijective function $g : \mathbf{T} \to \mathbb{N}$ and the inverse function $g^{-1} : \mathbb{N} \to \mathbf{T}$. We call $g\,t$ the **index of** t sometimes.

The idea here is simple to explain:

We construct a cumulative sequence T_n of duplicate free lists of terms such that $T_{n+1} = T_n \mathbin{+\!\!+} B$. We need to make sure that for every term s we can find an index n such that T_n contains s. If so, we use the position of s in T_n as the index of s. If we want to find the term belonging to a given index m (that is: $g^{-1}\ m$) we simply build the lists until an n such that T_n is big enough and take the term at the m-th position.

Since the lists are cumulative and duplicate free both operations are indeed functions that invert each other.

It remains to find such a list sequence T_n. We use a simple approach that is generalizable to arbitrary countable constructor types.

The idea gets described well by the following inference rules concerning membership in the single lists:

$$\frac{}{n \in T_n} \qquad \frac{s \in T_n \quad t \in T_n}{s\,t \in T_{n+1}} \qquad \frac{s \in T_n}{\lambda s \in T_{n+1}}$$

We will use the notation $A[n]$ for the n-th element in the list A. The function $\boldsymbol{\lambda} A\ n.\ A[n]$ is of type $\mathbf{T}$ *list* $\to \mathbb{N} \to \mathbf{T}_\perp$.

The definition of T is straightforward using the rules above. For $n = 0$ we can simply set $T_0 := [0]$. We need to make sure that T_{n+1} contains the variable $n+1$. Then, for every s we take λs to T_{n+1} if it was not contained in T_n. We do the same for $s\ t$ where both s and t were in T_n, again only if $s\ t$ was not contained in T_n already.

$$\begin{aligned} [\,] \times_{\mathit{app}} B &= [\,] \\ a :: A \times_{\mathit{app}} B &= A \times_{\mathit{app}} B +\!\!+ \mathit{map}\ (\boldsymbol{\lambda} x.\ ax)\ B \end{aligned}$$

$$\begin{aligned} T_0 &= [0] \\ T_{n+1} &= T_n +\!\!+ [n+1] +\!\!+ \mathit{filter}\ (\lambda x.x \notin T_n)\ (T_n \times_{\mathit{app}} T_n +\!\!+ \mathit{map}\ \lambda\ T_n) \end{aligned}$$

We obviously have:

Fact 9.2 *For $a \in A$ and $b \in B$ we have $a\ b \in A \times_{\mathit{app}} B$*

Four useful facts about T_n are easy to prove and follow from each other:

Fact 9.3

1. $\exists x, B. T_{n+1} = T_n +\!\!+ x :: B$
2. $T_n[m] = s \to T_{n+1}[m] = s$
3. $n_2 \geq n_1 \to T_{n_2}[m] = s \to T_{n_1}[m] = s$
4. $n_2 \geq n_1 \to s \in T_{n_1}[m] \to s \in T_{n_2}[m]$

We can prove that T fulfills the specification we gave above.

Lemma 9.4 $n \in T_n$

Proof By case analysis over n. ■

Lemma 9.5 $s \in T_n \to t \in T_n \to s\ t \in T_{n+1}$

Proof We want to prove that $s\ t \in T_n +\!\!+ [n] +\!\!+ \mathit{filter}\ (\lambda x.x \notin T_n)\ (T_n \times_{\mathit{app}} T_n + + \mathit{map}\ \lambda\ T_n)$. If $s\ t \in T_n$, then we are already done. If not, we already have shown that $s\ t \in T_n \times_{\mathit{app}} T_n$ and because $s\ t \notin T_n$, we know that $s\ t \in \mathit{filter}\ (\lambda x.x \notin T_n)\ (T_n \times_{\mathit{app}} T_n)$ and thus $s\ t \in T_{n+1}$. ■

Lemma 9.6 $s \in T_n \to \lambda s \in T_{n+1}$

Proof Parallel to the one above. ∎

We now define an alternative size-function for our terms and show that T_n contains all terms with size less or equal than n:

$$\begin{aligned}|n| &= n\\ |s\,t| &= 1 + |s| + |t|\\ |\lambda s| &= 1 + |s|\end{aligned}$$

Note that in difference to our previous size function the size of a variable is not 1, but the de Bruijn index of the variable. We are able to show the following lemma.

Lemma 9.7 $s \in T_{|s|}$

Proof By induction over s.

If s is a variable, we are done, because $n \in T_n$ holds.

If $s = s\ t$, and $s \in T_{|s|}$, $t \in T_{|t|}$, then by the lemma above it is enough to show that $s, t \in T_{1+|s|+|t|}$, which follows point (4) of Fact 9.3.

If $s = \lambda s$ and $s \in T_{|s|}$, then by the correctness of *map* it is enough to show that $\lambda s \in T_{1+|s|}$, which holds by Lemma 9.6. ∎

For the surjectivity of g we also need the result:

Lemma 9.8 $|T_n| > n$

Proof By induction over n. ∎

This gives us the ability to define

$$\begin{aligned}g &:= \boldsymbol{\lambda} s.\ \textit{pos}\ s\ T_{|s|}\\ g^{-1} &:= \boldsymbol{\lambda} s.\ T_n[n]\end{aligned}$$

We can show that $g^{-1}(g\ s) = s$ which gives us injectivity of g and surjectivity of g^{-1}.

Theorem 9.9 (Injectivity of the Enumeration) $g^{-1}(g\ s) = s$

Proof We start with a case analysis over $g\ s$, that is over the position of s in $T_{|s|}$. We know that this is defined by Lemma 9.7, thus we can assume that $T_{|s|}[n] = s$. We then analyze $g^{-1}\ n$, which is $T_n[n]$. By Lemma 9.8 we know, that this is also defined, so we assume that $T_n[n] = t$. It remains to show that $s = t$.

$|s| = n$: Then we have nothing to prove, because there is just one element at position n.

$|s| > n$: Then we know by point 3 in Fact 9.3 that $T_{|s|}[n] = t$ and we are done.

$|s| < n$: Again, by point 3 in Fact 9.3 it follows that $T_n[n] = s$. ■

The proof that g is also surjective using $g(g^{-1}\ n) = n$ takes a bit more effort. For this we need the result that T_n is a duplicate-free list. We first prove two auxiliary lemmas:

Lemma 9.10 $m > n \rightarrow m \notin T_n$

Proof By induction over n, case analysis over m and the fact that all elements of $A \times_{app} B$ are applications. ■

Lemma 9.11 *T_n is duplicate free.*

Proof We use induction over n. The base case is trivial. For the inductive step, assume that T_n is duplicate free.

We then show that T_{n+1} also contains no duplicates. Because $T_{n+1} = T_n +\!\!+ [n] + +$ *filter* $(\boldsymbol{\lambda} x.\ x \notin T_n)$ $(T_n \times_{app} T_n +\!\!+$ *map* $\lambda\ T_n)$, it is enough to show that T_n and *filter* $(\lambda x.x \notin T_n)$ $(T_n \times_{app} T_n +\!\!+$ *map* $\lambda\ T_n)$ have no common elements and that both are duplicate free. The second part (duplicate-freeness of T_n) follows from the inductive hypothesis.

The first part uses the previous lemma and the correctness of filter and is straightforward. The last part is the interesting one.

We want to show that*filter* $(\boldsymbol{\lambda} x.\ x \notin T_n)$ $(T_n \times_{app} T_n +\!\!+$ *map* $\lambda\ T_n)$ does not contain any duplicates if T_n does not. We first need to show that the variable $n + 1$ is neither in $T_n \times_{app} T_n$ nor in *map* $\lambda\ T_n$. Both facts hold, because $A \times_{app} B$ contains only applications and the mapped part only λ-terms. Thus the duplicate freeness of *filter* $(\lambda x.x \notin T_n)$ $(T_n \times_{app} T_n +\!\!+$ *map* $\lambda\ T_n)$ remains.

For this it is enough to show that both $T_n \times_{app} T_n$ and *map* $\lambda\ T_n$ are dupfree. The first one is provable by a general lemma about $\times_{app}$, the second one is a property of *map*, because T_n is dupfree by assumption and the function which puts a λ above its argument is clearly injective. ■

It follows that

Theorem 9.12 (Surjectivity of the Enumeration) $g\ (g^{-1}\ n) = n$

Proof The proof is analogous to Theorem 9.9. ■

We have an explicit, computable bijection g between $\mathbf{T}$ and $\mathbb{N}$. Our next step is to internalize its inverse g^{-1} into a combinator U. To do so we need to internalize the functions on lists we used.

9.3 A List Libray for *L*

In the proof of a bijection between $\mathbf{T}$ and $\mathbb{N}$ we heavily used List functions in Coq. When translating the functions g and g^{-1} in the usual way to procedures in L, we also need this list functions. We internalize lists over an arbitrary Scott-encodable data type X using Scott-encoding again. Thus we only need to verify that there are procedures internalizing *cons*, *append*, *map*, *membership*, *filter*, *nth* and *pos*.

Some of them, like *cons*, *append* and *nil* are trivial to internalize to combinators *Nil*, *Append* and *Cons*.

Some others like *Map*, *El* and *Filter* need more thought.

Fact 9.13 *There is a combinator Map such that for every procedure u:*

1. $\mathit{Map}\ u\ \mathit{Nil} \equiv \mathit{Nil}$
2. $\mathit{Map}\ u\ \ulcorner a :: A\urcorner \equiv \mathit{Cons}\ (u\ulcorner a\urcorner)((\lambda x.\mathit{Map}\ x)u\ulcorner A\urcorner)$

Lemma 9.14 *If u internalizes f, then $\mathit{Map}\ u\ \ulcorner A\urcorner \equiv \ulcorner \mathit{map}\ f\ A\urcorner$*

Proof By induction over A using the two properties above. ■

To show that there is a combinator *El* internalizing membership we assume that there is a combinator *Eq* which decides equality on the assumed type X.

First we show the two recursive equations:

Fact 9.15 *If $\forall xy : X. x = y \wedge \mathit{Eq}\ \ulcorner x\urcorner\ulcorner y\urcorner \equiv \mathit{true} \vee x \neq y \wedge \mathit{Eq}\ \ulcorner x\urcorner\ulcorner y\urcorner \equiv \mathit{false}$ then there is a combinator El such that:*

1. $\mathit{El}\ \ulcorner s\urcorner\ \mathit{Nil} \equiv false$
2. $\mathit{El}\ \ulcorner s\urcorner\ulcorner a :: A\urcorner \equiv \mathit{orelse}\ (\mathit{Eq}\ \ulcorner a\urcorner\ulcorner s\urcorner)\ ((\lambda x.\mathit{El}\ x)\ \ulcorner s\urcorner\ulcorner A\urcorner)$

Then the correctness of *El* follows:

Lemma 9.16 *For Eq as above we have: $\mathit{El}\ \ulcorner s\urcorner\ulcorner A\urcorner \equiv \mathit{true} \wedge s \in A \vee \mathit{El}\ \ulcorner s\urcorner\ulcorner A\urcorner \equiv \mathit{false} \wedge s \notin A$*

Proof By induction over A, the two properties above and the correctness of *Eq*. ■

The correctness lemma for *Filter* is stated as follows:

Lemma 9.17 *Let u be a decider for $P : X \to$ **Prop**. Then Filter $u \ulcorner A \urcorner \equiv \ulcorner$filter $P\ A \urcorner$.*

Proof By induction over A. The base case is trivial. In the inductive step one first needs to use the fact that u is a decider, then decide if p holds on the first element of the list afterwards. Two cases generate a direct contradiction, the two others are straightforward. ∎

The last interesting procedure we internalize is *pos*. There is no function in the Coq library which gives the index of the first occurrence of an element in a list. Thus we first define:

```
Fixpoint pos (X : Type) {e : eq_dec X} (s : X) (A : list X) :=
match A with
| nil ⇒ None
| a :: A ⇒ if decision (s = a) then Some 0 else match pos s A with None ⇒ None
                                                                | Some n ⇒ Some (S n) end
end.
```

Fact 9.18 *There is a combinator Pos for which the following two equivalences hold:*

$$\begin{aligned} Pos \ulcorner s \urcorner\ Nil &\equiv None \\ Pos \ulcorner s \urcorner \ulcorner a :: A \urcorner &\equiv Eq \ulcorner s \urcorner \ulcorner a \urcorner\ (\lambda(Some\ Zero)) \\ &\qquad (\lambda((\lambda x.Pos\ x) \ulcorner s \urcorner \ulcorner A \urcorner)(\lambda n.Some\ (Succ\ n))\ None))I \end{aligned}$$

We can prove that:

Lemma 9.19 $Pos \ulcorner s \urcorner \ulcorner A \urcorner \equiv None \land pos\ s\ A = \bot \lor Pos \ulcorner s \urcorner \ulcorner A \urcorner \equiv Some\ \overline{n} \land pos\ s\ A = \lfloor n \rfloor$

Proof By induction over A. ∎

9.4 Internalized Enumeration of Terms

The last thin we need to do to build an enumerator is to internalize the functions used in Section 9.2. We use the well-known technique to internalize the $\times_{app}$, $|\cdot|$ and T_n procedures. Thus we define combinators *AppCross*, *ESize* and *TT* such that:

1. $AppCross\ Nil \ulcorner B \urcorner \equiv Nil$
2. $AppCross \ulcorner a :: A \urcorner \ulcorner B \urcorner \equiv Append\ (AppCross\ A\ B)\ (Map\ (\lambda x.(App \ulcorner a \urcorner)\ x) \ulcorner B \urcorner)$
3. $ESize \ulcorner n \urcorner \equiv \overline{n}$
4. $ESize \ulcorner st \urcorner \equiv Add\ (Add\ \overline{1}\ ((\lambda x.ESize\ x) \ulcorner s \urcorner))((\lambda x.ESize\ x) \ulcorner t \urcorner)$
5. $ESize \ulcorner \lambda s \urcorner \equiv Succ\ ((\lambda x.ESize\ x) \ulcorner s \urcorner)$

The equations which need to hold for the combinator TT are the obvious ones.

Lemma 9.20 *AppCross* $\ulcorner A \urcorner \ulcorner B \urcorner \equiv \ulcorner A \times_{app} B \urcorner$

Proof By induction over A, the two equations above and the correctness of *Map* and *Append*. ■

Lemma 9.21 *ESize* $\ulcorner s \urcorner \equiv \overline{|s|}$

Proof By induction over s, the three equations above and the correctness of *Add* and *Succ*. ■

Lemma 9.22 $TT\ \overline{n} \equiv \ulcorner T_n \urcorner$

Proof By induction over n and all the correctness results above. ■

The last step is a combinator U internalizing the function g^{-1} we were searching for:

$$U := \lambda n.Nth\ (TT\ n)\ n)\ I\ \ulcorner 0 \urcorner$$

Lemma 9.23 $U\ \overline{n} \equiv \ulcorner g^{-1} n \urcorner$

Proof By the correctness of TT and Nth it is enough to prove that:

1. $T_n[n] = \lfloor t \rfloor$, then $(\lambda sn.s\ \ulcorner t \urcorner)\ I\ \ulcorner 0 \urcorner \equiv \ulcorner t \urcorner$, which holds
2. $T_n[n] = None$, then $(\lambda sn.n)\ I\ \ulcorner 0 \urcorner \equiv \ulcorner 0 \urcorner$, which also holds. ■

9.5 Equivalence of L-Enumerability and L-Acceptability

9.5.1 L-Enumerability implies L-Acceptability

The intuitive idea behind this proof is as follows: We know that the predicate P : $\mathbf{T} \to \mathbf{Prop}$ is L-enumerable, that is we have an enumerator F. We need to build an acceptor u, that halts on $\ulcorner s \urcorner$ if and only if Ps holds. u simply enumerates the terms satisfying P using F and does a linear search for s as a result. If F returns s, it can converge and indeed Ps holds. If F never converges to *Some* s, u will diverge.

We build a combinator Re that takes $\overline{n}$ and $\ulcorner s \urcorner$. It checks if $F\ \overline{n} \equiv$ *Some* s, converges if so and proceeds with $Re\ \overline{Sn}\ \ulcorner s \urcorner$ if not. Then $Re\ \overline{0}\ \ulcorner s \urcorner$ converges if and only if there is n with $F\ \overline{n} \equiv$ *Some* $\ulcorner s \urcorner$.

To build and verify Re we first fix a combinator F with $\forall n.F\ \overline{n} \equiv None \lor \exists s.F\ \overline{n} \equiv$ *Some* $\ulcorner s \urcorner$. Note that a combinator F with weaker properties than enumerators have already suffices.

Fact 9.24 *There is a combinator Re such that*

$$Re\,\overline{n}\,\ulcorner t\urcorner \equiv F\,\overline{n}\,(\lambda s.Eq\,s\,\ulcorner t\urcorner\,I\,(\lambda\lambda(\lambda x.Re\,x)\,(Succ\,\overline{n})\,\ulcorner t\urcorner)\,I)(\lambda(\lambda x.Re\,x)\,(Succ\,\overline{n})\,\ulcorner t\urcorner)\,I$$

Re is antimonotonic in the sense that if $Re\,\overline{n}\,\ulcorner s\urcorner$ converges it also does so for every index $m \leq n$.

Lemma 9.25 $Re\,\overline{Sn}\,\ulcorner s\urcorner \Downarrow \rightarrow Re\,\overline{n}\,\ulcorner s\urcorner \Downarrow$

Proof Assume $Re\,\overline{Sn}\,\ulcorner s\urcorner$ converges with some value v. Analyse what $F\,\overline{n}$ evaluates to. This is either *None* or *Some* t. In any case we need to show that $Re\,\overline{n}\,\ulcorner s\urcorner$ converges.

We have $Re\,\overline{n}\,\ulcorner s\urcorner \equiv Re\,\overline{Sn}\,\ulcorner s\urcorner \equiv v$ in the first case. For the second case we either have $s = t$ and $Re\,\overline{n}\,\ulcorner s\urcorner$ converges directly. Or, $s \neq t$ and again $Re\,\overline{n}\,\ulcorner s\urcorner \equiv Re\,\overline{Sn}\,\ulcorner s\urcorner \equiv v$. ∎

Lemma 9.26 $n \geq m \rightarrow Re\,\overline{n}\,\ulcorner s\urcorner \Downarrow \rightarrow Re\,\overline{m}\,\ulcorner s\urcorner \Downarrow$

Proof Follows directly from 9.25. ∎

Lemma 9.27 $Re\,\overline{n}\,\ulcorner s\urcorner \Downarrow \rightarrow \exists n.F\,\overline{n} \equiv \mathit{Some}\,\ulcorner s\urcorner$

Proof Since Re is not definable in Coq directly, we again use the technique of doing induction over the length of the reduction path.

Assume that $Re\,\overline{n}\,\ulcorner s\urcorner \succ^k v$. By complete induction over k and the statement $\forall n.Re\,\overline{n}\,\ulcorner s\urcorner \succ^k v \rightarrow \exists n.F\,\overline{n} \equiv \mathit{Some}\,\ulcorner s\urcorner$ we are allowed to assume the statement for $k' < k$.

The proof uses a case analysis over the value of $F\,\overline{n}$.

1. If $F\,\overline{n} \equiv \mathit{None}$ we know that there is $m > 0$ such that $Re\,\overline{n}\,\ulcorner s\urcorner \succ^m Re\,\overline{Sn}\,\ulcorner s\urcorner$. Thus there is k' with $Re\,\overline{Sn}\,\ulcorner s\urcorner \succ^{k'} v$ and we apply the inductive hypothesis.
2. If $F\,\overline{n} \equiv \mathit{Some}\,\ulcorner s\urcorner$ we are done.
3. If $F\,\overline{n} \equiv \mathit{Some}\,\ulcorner t\urcorner$ we have in parallel to the first case $m > 0$ such that $Re\,\overline{n}\,\ulcorner s\urcorner \succ^m Re\,\overline{Sn}\,\ulcorner s\urcorner$ and thus k' with $Re\,\overline{Sn}\,\ulcorner s\urcorner \succ^{k'} v$. Using the inductive hypothesis we are done. ∎

Lemma 9.28 *If P is L-enumerable it is L-acceptable.*

Proof Assume there is an enumerator F. Then $u := (\lambda x.Re\,\overline{0}\,x)$ where using F in the implementation of Re as shown above is an acceptor for P. We show two directions, namely that $u\,\ulcorner t\urcorner$ converges if $P\,t$ holds and the converse.

For the first direction assume that $P\ t$ holds. Then there is n with $F\ \overline{n} \equiv \mathit{Some}\ \ulcorner t \urcorner$. Using this we know that $\mathit{Re}\ \overline{n}\ \ulcorner t \urcorner$ converges. We have to show that $\mathit{Re}\ \overline{0}\ \ulcorner t \urcorner$ converges, which follows with Lemma 9.25.

For the second direction assume that u converges on $\ulcorner t \urcorner$, that means $\mathit{Re}\ \overline{0}\ \ulcorner t \urcorner$ converges. Since F fulfills the properties outlined in Section 9.5.1 we know that there is n with $F\ \overline{n} \equiv \mathit{Some}\ \ulcorner t \urcorner$ by Lemma 9.27. Everything we need to prove here is that Ps holds. We exploit the totality of F. Either we have $F\ \overline{n} \equiv \mathit{None}$, which is a contradiction. From $F\ \overline{n} \equiv \mathit{Some}\ \ulcorner t' \urcorner \land Pt'$ we know that $t = t'$ and thus Pt holds.■

9.5.2 *L*-Acceptability implies *L*-Enumerability

To construct an enumerator from an acceptor we again employ a well-known idea of classical computability theory. An intuituive description of the idea is as follows: We enumerate all terms in a row using the ideas from Section 9.2. The enumerator at index $\overline{1}$ simply returns the value of the first term evaluated one step using the internalized step-indexed self-interpreter from Chapter 6. At index $\overline{2}$, it evaluates the first term two steps. At index $\overline{3}$, it evaluates the second term one step. Then it proceeds like this: Evaluate the first term three steps, the second term two steps, the third term one step and so on.

In fact, we can even go one step further: We do not need to do this in a diagonal order, the order does not even matter at all, as long as we reach every term with every index eventually. Thus, we use an internalized surjection $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$. We use the first component as the index of the term and the second one as the evaluation bound. We assume that there is a surjective function $c : \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ and a combinator C that internalizes it. This is enough to prove that semi-decidability implies recursive enumerability. We explicitly construct and verify such a surjection c in Section 9.5.2.

Theorem 9.29 *Every L-acceptable predicate is L-enumerable.*

Proof Let P be a predicate and u an acceptor for P. We define F as

$$\lambda n. C\ n\ (\lambda n_1 n_2. \mathit{Eva}\ n_1\ (\mathit{App}\ \ulcorner u \urcorner\ (Q\ (U\ n_2)))(\lambda\ \mathit{Some}\ (U\ n_2))\ \mathit{None})$$

. F is an enumerator for P.

We need to show two parts:

- $\forall n. F\ \overline{n} \equiv \mathit{none} \lor \exists s. F\ \overline{n} \equiv \mathit{Some}\ \ulcorner s \urcorner \land Ps$
  Let n be a natural number and assume $c\, n = (n_1, n_2)$. Examine the evaluation of $u\ \ulcorner g^{-1}\ n_2 \urcorner$ evaluates in n_1 steps.

If $u \ulcorner g^{-1}\, n_2 \urcorner \Downarrow^{n_1} v$ the right part of the disjunction holds. We have:

$$\begin{aligned} F\,\overline{n} &\equiv \mathit{Eva}\,\overline{n_1}\;(\mathit{App}\ulcorner u \urcorner\;(Q\;(U\;\overline{n_2})))(\lambda\;\mathit{Some}\;(U\;\overline{n_2}))\;\mathit{None} \\ &\equiv \mathit{Eva}\,\overline{n_1}\ulcorner u \ulcorner g^{-1}\;n_2 \urcorner\urcorner(\lambda\;\mathit{Some}\;(U\;\overline{n_2}))\;\mathit{None} \\ &\equiv \mathit{Some}\;(U\;\overline{n_2}) \\ &\equiv \mathit{Some}\;(\ulcorner g^{-1}\;n_2 \urcorner) \end{aligned}$$

It remains to show that $P(g^{-1}\,n_2)$ also holds. But this is the case, since $u \ulcorner g^{-1}\,n_2 \urcorner \Downarrow^{n_1} v$ and thus the acceptor u does obviously converge on $\ulcorner g^{-1}\,n_2 \urcorner$.

If we do not have a value v with $u \ulcorner g^{-1}\,n_2 \urcorner \Downarrow^{n_1} v$ it follows that:

$$\begin{aligned} F\,\overline{n} &\equiv \mathit{Eva}\,\overline{n_1}\;(\mathit{App}\ulcorner u \urcorner\;(Q\;(U\;\overline{n_2})))(\lambda\;\mathit{Some}\;(U\;\overline{n_2}))\;\mathit{None} \\ &\equiv \mathit{Eva}\,\overline{n_1}\ulcorner u \ulcorner g^{-1}\;n_2 \urcorner\urcorner(\lambda\;\mathit{Some}\;(U\;\overline{n_2}))\;\mathit{None} \\ &\equiv \mathit{None} \end{aligned}$$

This shows the left side of the disjunction.

- $\forall s. P s \to \exists n. F\,\overline{n} \equiv \mathit{Some}\ulcorner s \urcorner$
  The idea here is easy. Since s satisfies P, u converges on s in n steps. We simply need the index m of the pair $(n, g\;s)$, this directly yields:

$$\begin{aligned} F\,\overline{m} &\equiv \mathit{Eva}\,\overline{n}\;(\mathit{App}\ulcorner u \urcorner\;(Q\;(U\;\overline{g\;s})))(\lambda\;\mathit{Some}\;(U\;\overline{g\;s}))\;\mathit{none} \\ &\equiv \ulcorner \mathit{eva}\;n\;(u \ulcorner s \urcorner)\urcorner(\lambda\;\mathit{Some}\;(U\;\overline{g\;s}))\;\mathit{none} \\ &\equiv \mathit{Some}\;(U\;\overline{g\;s}) \\ &\equiv \mathit{Some}\ulcorner s \urcorner \end{aligned}$$

■

A Pairing Function

We reuse the idea from Section 9.2 to get a surjection from $\mathbb{N}$ to $\mathbb{N} \times \mathbb{N}$. Everything we need to do is to define cumulative lists C_n such that pair of natural numbers occurs eventually in C_n for some n. The function then just maps a natural number m to the pair at position m in this cumulative lists. They key idea is:

$$\frac{}{(n,0) \in T_n} \qquad \frac{(n,m) \in T_n}{(n,m+1) \in T_{n+1}}$$

In parallel to the function T from Section 9.2 we define:

$$\begin{aligned} C_0 &= [(0,0)] \\ C_{n+1} &= C_n +\!\!+ [(n+1,0)] +\!\!+ \mathit{filter}\;(\lambda x. x \notin C_n)\;(\mathit{map}\;(\lambda(n,m).(n,m+1))\;C_n) \end{aligned}$$

We then have:

Fact 9.30

- $\exists x, B. C_{n+1} = C_n :: x :: B$
- $C_n[m] = s \rightarrow C_{n+1}[m] = s$
- $n_2 \geq n_1 \rightarrow C_{n_2}[m] = s \rightarrow C_{n_1}[m] = s$

We define the size of a pair (n, m) as $|(n, m)| = 1 + n + m$. Then we have two important lemmas:

Lemma 9.31 $(n, m) \in C_{|(n,m)|}$.

Proof By induction over n. Every case is easy. ■

Lemma 9.32 $|C_n| > n$

Proof By induction over n. Every case is trivial again. ■

We define the pairing function c and its inverse c^{-1}:

$$c\, n := T_n[n]$$
$$c^{-1}\,(n, m) := \mathit{pos}\,(n, m)\; C_{|(n,m)|}$$

We prove that c is surjective, which follows since c^{-1} is a right inverse to c. In fact, it is even bijective (because c^{-1} is a left inverse), but we leave this fact open here, since it is not needed.

Theorem 9.33 (Surjectivity of the pairing function) $c\,(c^{-1}\,(n, m)) = (n, m)$

Proof The proof is analogous to Lemma 9.9. ■

Corollary 9.34 $c : \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ *is surjective.*

The internalization of c to a combinator C is trivial. We use exactly the same method as in Section 9.2.

9.6 The DA Theorem

Using the possibility to enumerate terms we are now able to prove that existential quantification over a L-decidable predicate is L-acceptable.

We define a term *Ex* that performs unbounded search on all natural numbers. We then internalize our enumeration of terms to prove the wanted result. For *Ex* it should hold that:

$$Ex\,\overline{n}\,v \equiv v\,\overline{n}\,(\lambda I)\,(\lambda\,Ex\,(Succ\,\overline{n})\,v)\,I$$

We first prove the following results:

Fact 9.35

- $\forall v.\mathbf{P}v \to\ v\,\overline{n} \equiv true \to\ Ex\,\overline{n}\,v \Downarrow$
- *If v is a procedure, such that $v\,\overline{n}$ always evaluates to true or false, then*
  - $Ex\,\overline{Sn}\,v \Downarrow \to Ex\,\overline{n}\,v \Downarrow$
  - *If $n \geq m$ then $Ex\,\overline{n}\,v \Downarrow \to Ex\,\overline{m}\,v \Downarrow$*

The first statement is trivial to prove. The second one uses a case distinction over the value of $v\,\overline{n}$ and is straight forward. The third follows directly from the second.

It remains that:

Lemma 9.36 *If v is a procedure, such that $v\,\overline{n}$ always evaluates to true or false and $Ex\,\overline{n}\,v \Downarrow$ then there is $k : \mathbb{N}$ such that $v\,\overline{k} \equiv true$*

Proof Assume $Ex\,\overline{n}\,v$ converges. Let l be the length of the reduction path. By size induction we are able to assume that if $Ex\,\overline{m}\,v$ converges in less than l steps, then there is $k' : \mathbb{N}$ such that $v\,\overline{k}' \equiv true$.

Consider the evaluation of $v\,\overline{n}$. If it evaluates to *true*, then choose k to be n. If not, then we know that $Ex\,\overline{n}\,v \succ^l Ex\,\overline{Sn}\,v$ where $l > 0$. Then $Ex\,\overline{n}\,v$ also converges, but in less than l steps and the result follows by the inductive hypothesis. ∎

Lemma 9.37 *If P is an L-decidable predicate on natural numbers, then the predicate $\boldsymbol{\lambda}t.\ \exists n.Pn$ is semi-decidable.*

Proof Let v be a decider for P. Then $\lambda x.Ex\,\overline{0}\,v$ is the wanted acceptor.

We actually need to prove:

$$(\exists n.Pn)\ \leftrightarrow\ Ex\,v \Downarrow$$

The direction from left to right follows from the first statement in Fact 9.35 above. The direction from right to left is the last lemma. ∎

Recall the enumeration for terms consisting of the two functions $g : \mathbf{T} \to \mathbb{N}$ and $g^{-1} : \mathbb{N} \to \mathbf{T}$ with the equations $g^{-1}(g\ s) = s$.

Fact 9.38 *There is a combinator U fulfilling $U\,\overline{n} \equiv \ulcorner g^{-1}\,n \urcorner$.*

Now we can prove:

Theorem 9.39 (DA) *If a predicate P is L-decidable, the predicate $\boldsymbol{\lambda}.\ \exists s. Ps$ is acceptable.*

Proof Let v be a decider for P. We first show that $\boldsymbol{\lambda} n : \mathbb{N}.\ P(g^{-1}\ n)$ is a decidable predicate on natural numbers.

Consider the procedure $v := \lambda x.v\ (U\ x)$. $v\,\overline{n} \equiv v\ (U\ \overline{n}) \equiv v\,\ulcorner g^{-1}\ n \urcorner$ holds. $v\,\ulcorner g^{-1}\ n \urcorner$ is either equivalent to *true* or *false*, which proves what we wanted.

Thus by the lemma above we know that the predicate $\boldsymbol{\lambda}.\ \exists n : \mathbb{N}.P(g^{-1}n)$ is acceptable and

$$\pi\ v\ t \leftrightarrow \exists n : \mathbb{N}.P(g^{-1}\ n)$$

But if $\exists n : \mathbb{N}.P(g^{-1}\ n)$, then we already have a term satisfying P. On the other hand, if we have a term s satisfying P, then $g^{-1}\ (g\ s)$ satisfies M, because $g^{-1}\ (g\ s) = s$. Thus $(\exists n : \mathbb{N}.P(g^{-1}\ n)) \leftrightarrow \exists s : \mathbf{T}.P\ s$.

We then conclude that

$$\pi\ v\ t \leftrightarrow \exists s : \mathbf{T}.P\ s$$

and by this the predicate $\boldsymbol{\lambda}.\ \exists s : \mathbf{T}.P\ s$ is semi-decidable with acceptor v. ■

Chapter 10

More on L-Acceptability

In Section 5.3 we showed that L-acceptable predicates are closed under conjunction. Several questions came up during the development of CCT that could only be answered with a certain machinery, for instance if L-acceptable predicates are also closed under disjunction. We have built a considerable framework, namely a lot of theorems on criteria for the L-undecidability or L-unacceptablity of predicates and, maybe most important, a self-interpreter. We are now able to answer the last open questions.

In particular, we will show that L-acceptable predicates are not closed under complement. Then, we can show that L-acceptable predicates are closed under disjunction. Finally we show that there are predicates that are neither L-acceptable nor L-coacceptable.

10.1 Semi-Decidability of the Self Halting Predicate

In Lemma 4.3 we already saw that the complement of the self-halting predicate $\lambda t.\ t \ulcorner t \urcorner \Downarrow$ is L-undecidable.

Lemma 10.1 *The self-halting predicate is L-acceptable.*

Proof The term $u := \lambda t.Eval(App\ t\ (Q\ t)$ is an acceptor, since:

$u \ulcorner t \urcorner \equiv Eval(App \ulcorner t \urcorner\ (Q \ulcorner t \urcorner) \equiv Eval \ulcorner t \ulcorner t \urcorner \urcorner$.

Since $Eval \ulcorner t \ulcorner t \urcorner \urcorner \Downarrow\ \leftrightarrow t \ulcorner t \urcorner \Downarrow$ we are done.

10.2 L-Acceptable Predicates are closed under Disjunction

If we want to show that L-acceptable predicates are closed under disjunction we use an already well-known idea again: We take two acceptors u and v of two L-

acceptable predicates P and Q and under input $\ulcorner s \urcorner$ interleave the execution of $u \ulcorner s \urcorner$ and $v \ulcorner s \urcorner$ via parallel-or.

Lemma 10.2 *L-acceptable predicates are closed under disjunction.*

Proof Let P and Q be L-acceptable predicates with acceptors u and v respectively. We claim that $w := \lambda t.\mathit{Por}(\mathit{App} \ulcorner u \urcorner (Q\, t))(\mathit{App} \ulcorner v \urcorner (Q\, t))$ is an acceptor for $\boldsymbol{\lambda} t.\ Pt \vee Qt$.

Notice that $w \ulcorner t \urcorner \equiv \mathit{Por} \ulcorner u \ulcorner t \urcorner \urcorner \ulcorner v \ulcorner t \urcorner \urcorner$.

Assume $P\ t\ \vee\ Q\ t$. Then we either have $u \ulcorner t \urcorner \Downarrow$, which directly yields the convergence of $w \ulcorner t \urcorner$ by the above equivalence and the correctness of *Por* (Lemma 8.4). Or we have $v \ulcorner t \urcorner \Downarrow$, so that we get the same result on the same way, just with Lemma 8.5.

For the other direction assume that $w \ulcorner t \urcorner$ converges. This directly yields that either u or v converges on $\ulcorner t \urcorner$ by Lemma 8.7. ∎

10.3 Hard Predicates

Until now we have shown several predicates to be L-unacceptable. We did not show the interesting result that there are predicates which are neither L-acceptable nor L-coacceptable. We call such predicates hard predicates.

An example for a hard predicate is totality of a term, that means convergence on all possible inputs. It is neither L-acceptable nor L-coacceptable. Since L-acceptability can be seen as the weakest notion of computability, the property of the convergence of a program under all inputs is not determinable in a general way, which has a lot of implications in the verification of real-world programs.

We split the proof in two parts:

Lemma 10.3 *The predicate $\boldsymbol{\lambda} t.\ \forall s. \pi\ t\ s$ is not L-acceptable.*

Proof Assume an acceptor u. Then we can give an acceptor for the complement of the self-halting problem. For this we define a term v_s, which is total if and only if $s \ulcorner s \urcorner$ converges:

$$v_s\ :=\ \lambda y.\ \mathit{Eva}\ (\mathit{Size}\ y) \ulcorner s \ulcorner s \urcorner \urcorner\ (\lambda \Omega)\ I$$

We show this as follows:

Assume $s \ulcorner s \urcorner$ does not converge. Then we know that there is no v with $s \ulcorner s \urcorner \Downarrow^{|t|} v$ and we find $v_s \ulcorner t \urcorner \equiv I$. For the other direction assume that $v_s \ulcorner t \urcorner$ converges for all t and also that $s \ulcorner s \urcorner$ converges in n steps to v. We know that $n > 1$ and that there is a term t with $|t| = n$ (because the size function is obviously surjective). Thus

$v_s \ulcorner t \urcorner \equiv \mathit{Eva}\, \overline{n}\, \ulcorner s \ulcorner s \urcorner \urcorner\, (\lambda\Omega)\ I \equiv \Omega$. But since v_s is total we find a procedure v' with $v_s \ulcorner t \urcorner \equiv v'$. We can show now that $\Omega \equiv v'$, which is for sure a contradiction:

$$v' \equiv v_s \ulcorner t \urcorner \equiv \Omega$$

We show that w is an acceptor for the self-halting problem. Note that w does nothing more than applying u to the encoded version of v_s.

$$\begin{aligned} w \ :=\ & \lambda x.u\, (\mathit{Lam}\, (\mathit{App}\, (\mathit{App}\, (\mathit{App}\, (\mathit{App} \\ & \qquad \ulcorner \mathit{Eva} \urcorner \\ & \qquad (\mathit{App}\, \ulcorner \mathit{Size} \urcorner\, \ulcorner 0 \urcorner)) \\ & \qquad (\mathit{App}\ x\ (Qx))) \\ & \qquad \ulcorner \lambda\Omega \urcorner) \\ & \qquad \ulcorner I \urcorner)) \end{aligned}$$

Since for any term s we have $w\ulcorner s \urcorner \equiv u \ulcorner v \urcorner_s$ and $\pi\ s\ s\ \leftrightarrow\ v_s$ *is total* $\leftrightarrow\ \pi\ u\ v_s$ we know that w is an acceptor for the complement of the self-halting problem. Contradiction. ∎

Lemma 10.4 *The complement of the predicate* $\boldsymbol{\lambda} t.\ \forall s.\pi\ t\ s$*, that is* $\overline{T} := \boldsymbol{\lambda} s.\ \neg\forall t.\pi\ s\ t$ *is not L-acceptable.*

Proof We already know that the predicate $P := \boldsymbol{\lambda} s.\ \mathbf{P}s \land \forall t.\pi st$ is not L-acceptable by Corollary 5.15. With an acceptor u for $\overline{T}$, we could build an acceptor for P, since $\boldsymbol{\lambda} s.\ \mathbf{P}s$ is for sure L-acceptable. Contradiction. ∎

Theorem 10.5 *Totality of a term is neither L-acceptable nor L-coacceptable.*

Proof Follows from the last two lemmas. ∎

Appendix A

Coq Formalization

This thesis is accompanied by a Coq-Formalization.[1] In this chapter we will briefly discuss technical aspects of the formalization as well as the organization of the different files. Overall, the relevant part of the formalization has less than 1500 lines of proofs.

Relevant Coq-Techniques

There are several techniques that facilitated the definition of the semantics of L in Coq and the proofs.

First, it is possible to implicitly convert **term expressions** using named binders to the usual de Bruijn style terms. Another important helper is **setoid rewriting**, which can be used to shorten equivalence proofs and make them more natural. The tactic `crush` was implemented and used to prove trivial equivalences that follow by reduction. This tactic sometimes takes a lot of computation time, but shortens the proofs of trivial statements considerably. The last technique is a tactic that solves easy goals like closedness of a term or that a term is an abstraction.

We give an overview over those techniques in the following subsections.

Named Binders

The definition of terms in Coq is straightforward:

```
Inductive term : Type :=
  | var (n : ℕ) : term
  | app (s : term) (t : term) : term
  | lam (s : term).
```

Some notations can be used to make terms more readable:

[1]Available at `https://www.ps.uni-saarland.de/~forster/bachelor.php`

```
Coercion app : term ↣ Funclass.
Coercion var : ℕ↣ term.
Notation "(λ s )" := (lam s) (right associativity, at level 0).
```

The terms I and Ω then read λ 0 and (λ 0 0) (λ 0 0) respectively. This is good to read and not too hard to write for small terms, but it gets almost impossible for large terms.

To ease especially the definition of terms we defined **term expressions**, that use named binders and can be translated to terms easily.

Term expressions can be variables (represented by strings), applications and named binders. They embed ordinary terms via the notation ! t.

```
Inductive bterm : Type :=
| bvar (x : string) : bterm
| bapp (s t : bterm) : bterm
| blam (x : string) (s : bterm) : bterm
| bter (s : term) : bterm.

Coercion bvar : string ↣ bterm.
Coercion bapp : bterm ↣ Funclass.

Notation ".\ x , .. , y ; t" := ((blam x .. (blam y t) .. )) (at level 100, right associativity).
Notation "'λ' x , .. , y ; t" := ((blam x .. (blam y t) .. )) (at level 100, right associativity).

Notation "'!' s" := (bter s) (at level 0).
```

The conversion of term expressions to de Bruijn style terms is easy:

```
Fixpoint convert' (F : list string) (s : bterm) : term := match s with
| bvar x ⇒ match pos x F with None ⇒ #100 | Some t ⇒ # t end
| bapp s t ⇒ app (convert' F s) (convert' F t)
| blam x s ⇒ lam (convert' (x:: F) s)
| bter t ⇒ t
end.

Definition convert := convert' [].
```

To write term expressions wherever a term is expected we use a coercion again:

```
Coercion convert : bterm ↣ term.
```

Using this, one can define I, K and Ω like this:

```
Definition I : term := .\"x"; "x".

Definition K : term := .\"x","y"; "x".
Definition omega : term := .\"x"; "x" "x".
Definition Omega : term := omega omega.
```

Setoid-Rewriting

When doing equational proofs one simply rewrites expressions with equivalent ones. In Coq, this mechanism must first be enabled. We need to show that $\equiv$ is an equivalence relation and that $s \equiv s' \to\ t \equiv t' \to\ s\,t \equiv s'\,t'$ holds.

```
Instance equiv_Equivalence : Equivalence equiv.
Instance equiv_app_proper : Proper (equiv ==> equiv ==> equiv) app.
```

Rewriting with equations of the form $s \succ^* t$ should also be possible:

```
Instance star_PreOrder : PreOrder (star step).
```

Using this equivalences one can write a tactic that automatically rewrites the goal with every equivalence in the context:

```
Ltac rewrite_equiv :=
  match goal with
  | H : equiv ?s ?t ⊢_ ⇒ let H′ := stepsimpl_in H in rewrite H
  | H : star step ?s ?t ⊢_ ⇒ let H′ := stepsimpl_in H in rewrite H
end.
```

The tactic stempsimpl_in is used to simplify goals and is described in the next section.

Useful tactics

stepsimpl

In Coq one can use the **simpl** tactic to simplify goals. We have written a tactic stepsimpl that simplifies goals concerning terms of *L*. It unfolds all definitions and rewrites with registered equalities like those of the form $s^k_u = s$ for s closed.

```
Lemma proc_closed p : proc p → closed p.

Ltac rewrite_closed :=
match goal with
| H : proc _ ⊢_ ⇒ rewrite (proc_closed H)
| H : closed _ ⊢_ ⇒ rewrite H
end.

Ltac stepsimpl := cbv; autounfold with cbv; simpl;
                  autorewrite with cbv; repeat rewrite_closed.
```

The stepsimpl_in tactic works similarly, but allows the specification of an assumption where the simplification should be done.

crush

The tactic crush is able to deal with equivalences that follow by reduction only.

```
(* solving one−step reductions *)

Ltac tstep := stepsimpl;
 match goal with
   | [ ⊢step (app (lam _) (lam _) ) _ ] ⇒ eapply stepApp
   | [ ⊢step (app (lam _) _ ) _ ] ⇒ (eapply step_value; try eassumption;
                                          now eauto with cbv)
                                     | | eapply stepAppR
   | [ ⊢step _ _ ] ⇒ eapply stepAppL
end.

Ltac reduce := repeat tstep; stepsimpl.
```

The tstep tactic simply decides which constructor to use to prove the reduction step. If the left hand side of the reduction is an application of the form $(\lambda s)\ t$, then it first tries to use β-reduction by proving that t is a procedure.

The reduce tactic repeats this until a reduction is proven completely.

```
(* rewriting with equivalences in the context *)

Ltac rewrite_equiv :=
 match goal with
   | H : equiv ?s ?t ⊢_ ⇒ let H' := stepsimpl_in H in rewrite H
   | H : star step ?s ?t ⊢_ ⇒ let H' := stepsimpl_in H in rewrite H
end.
```

The rewrite_equiv tactic is able to rewrite with equivalences occuring in the assumptions.

```
(* solving equivalences by reduction *)

Ltac reduction' :=
 match goal with
   | [ ⊢equiv (app ?s _) (app ?s _) ] ⇒ eapply eqRef | | eapply equiv_trans_r
   | [ ⊢equiv _ _ ] ⇒ eapply eqRef | | (eapply eqTrans; [eapply eqStep; reduce | ])
   | [ ⊢star step (app ?s _) (app ?s _) ] ⇒ eapply starR | | eapply star_trans_r
   | [ ⊢star step _ _ ] ⇒ eapply starR | | (eapply starC; reduce) end; stepsimpl.

Ltac reduction := rewrite_equiv | | reduction'.
Ltac crush := stepsimpl; repeat reduction.
```

The crush tactic uses the reduction tactic repeatedly to solve equivalences. reduction rewrites with known equivalences. It then tries to solve the goal by the reflexivity of $\equiv$. If the goal is of the form $s\ t \equiv s\ t'$ it reduces this to $t \equiv t'$. In any other case, it uses the transitivity of $\equiv$ and tries to find the intermediate term via reduce.

Using crush takes a considerable amount of time and makes the compilation of the files quite slow, but shortens the proofs a lot.

value

The last tactic deals with goals like the closedness of a term or that a term is a procedure. One can register closedness-facts via Coq's Hint-mechanism.

```
Ltac value := try match goal with
    | [ ⊢proc _ ] ⇒ try eassumption; eauto with cbv; split; value
    | [ ⊢lambda _ ] ⇒ try eassumption; eauto with cbv; e∃ ; reflexivity
    | [ ⊢closed _ ] ⇒ try eassumption; eauto with cbv; intros k′ r′; simpl;
                              repeat rewrite_closed; autorewrite with cbv; reflexivity
end.
```

Organization of the Files

The Coq-Files of the accompanying formalization were compiled with Coq 8.4pl5. The easiest way to compile the whole formalization is by simply using the Makefile. The make-target `html` will check all proofs and produce a documentation containing all proofs in html-files.

The Programming Language

The definition of the terms of L is done in the file `Lvw.v`. The relations $\succ$ and $\equiv$ get defined there and all lemmas from chapter 2 can be found there.

Verification

The files `Nat.v` and `Bool.v` contain the definition of the natural numbers and booleans as well as the definition of the basic procedures. The encoding for terms can be found in `Encoding.v` and the definition and verification of the internalized size function in `Size.v`.

Decidability and Scott's Theorem

The definition of *L*-decidability can be found in `Decidability.v`. The two Fixed-Points theorems are in `Fixpoint.v` while Scott's Theorem is in `Scott.v`.

Acceptability and Rice's Theorem

The definition of semi-decidability is on `Acceptability.v`.

The internalized encoding combinators P and Q can be found in `Encoding.v`. Rice's Theorem can be found in `Rice.v`.

Step-Indexed Interpretation

The definition of the step-evaluation predicate as well as the eva function are in `Seval.v`.

Self-Interpretation

The self-interpreter is defined in `Eval.v`. The combinator for the definition of internalized partial functions is in `Partial.v`.

AD Theorem and Parallel Or

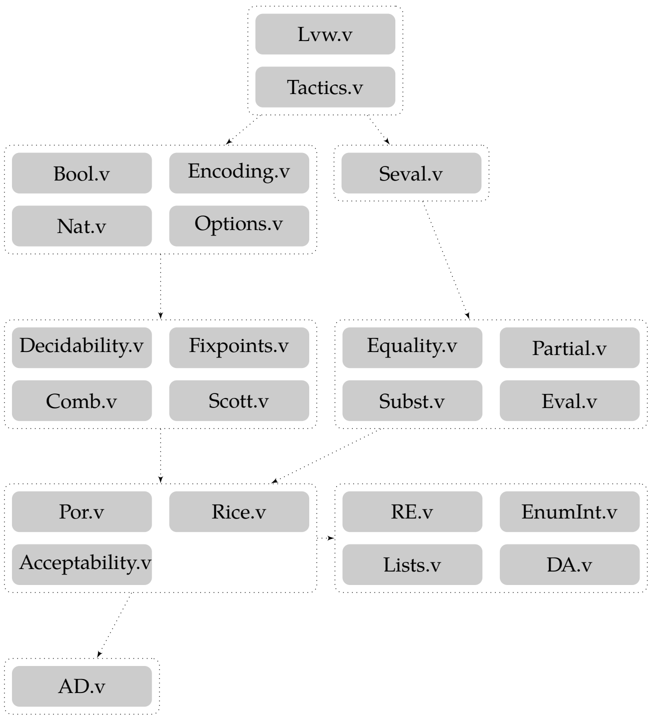
Parallel-or is defined and verified in `Por.v`. The AD-Theorem is shown in `AD.v`.

Recursive Enumerability

An enumeration for terms is defined in `Enum.v` and internalized in `EnumInt.v`. A list-library for *L* is given in `Lists.v`. The equivalence of *L*-acceptability and *L*-enumerability is shown in `RE.v`, the enumerator for pairs is given in `Pairs.v`. The DA Theorem is shown in `DA.v`.

More on *L*-Acceptability

Most results of this chapter are in `Acceptability.v`. The existence of hard predicates is shown in `MoreAcc.v`.



Structure of the Coq development

File	Spec.	Proofs
Acceptability.v	12	50
AD.v	2	16
Bool.v	20	12
Lvw.v	186	119
Computability.v	30	16
DA.v	16	71
Decidability.v	14	18
Encoding.v	48	72
Equality.v	49	17
Eval.v	73	73
Fixpoints.v	4	20
EnumInt.v	64	51
Enum.v	51	113
Lists.v	84	97
MoreAcc.v	4	62
Nat.v	39	36
Options.v	15	29
Partial.v	84	11
Por.v	55	48
Proc.v	43	100
RE.v	21	114
Rice.v	65	92
Scott.v	10	54
Seval.v	43	79
Size.v	6	13
Subst.v	6	13
Tactics.v	46	12
MoreAcc.v	4	62
In Total	1094	1470

Overview over Files

Bibliography

[1] Andrea Asperti and Wilmer Ricciotti. Formalizing Turing machines. In *Logic, Language, Information and Computation*, pages 1–25. Springer, 2012.

[2] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 2nd revised edition, 1984.

[3] George Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and Logic*. Cambridge University Press, 5th edition, 2007.

[4] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, pages 345–363, 1936.

[5] The Coq Development Team. *The Coq Reference Manual, version 8.4*, August 2012. Available electronically at `http://coq.inria.fr/doc`.

[6] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic: Volume II*. North-Holland Publishing Company, 1972.

[7] Ugo Dal Lago and Simone Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008.

[8] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[9] Jan Martin Jansen. Programming in the λ-calculus: From Church to Scott and back. In Peter Achten and Pieter Koopman, editors, *The Beauty of Functional Code*, volume 8106 of *LNCS*, pages 168–180. Springer Berlin Heidelberg, 2013.

[10] Neil D. Jones. *Computability and complexity from a programming perspective*. MIT Press, 1997.

[11] Dexter Kozen. *Automata and computability*. Undergraduate texts in computer science. Springer, 1997. ISBN 978-0-387-94907-9.

[12] Torben Æ. Mogensen. Efficient self-interpretations in lambda calculus. *J. Funct. Program.*, 2(3):345–363, 1992.

[13] Joachim Niehren. Functional computation as concurrent computation. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *POPL*, pages 333–343. ACM Press, 1996. ISBN 0-89791-769-3.

[14] Joachim Niehren. Uniform confluence in concurrent computation. *J. Funct. Program.*, 10(5):453–499, 2000.

[15] Michael Norrish. Mechanised computability theory. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *ITP*, volume 6898 of *LNCS*, pages 297–311. Springer, 2011. ISBN 978-3-642-22862-9.

[16] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[17] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[18] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. doi: 10.1112/plms/s2-42.1.230.

[19] Alan M. Turing. The þ-function in λ-k-conversion. *J. Symb. Log.*, 2(4):164, 1937. doi: 10.2307/2268281.

[20] Jian Xu, Xingyuan Zhang, and Christian Urban. Mechanising Turing machines and computability theory in Isabelle/HOL. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP*, volume 7998 of *LNCS*, pages 147–162. Springer, 2013.