A Coq Approach Towards SOS-Based Program Equivalence Proofs Using LCTRSs

Paul Reftu

September 10, 2020

Abstract

Program equivalence is an especially researched topic within the domain of formal software and hardware verification, often finding its use in solving problems such as program verification and optimization, algorithm recognition, regression testing, and more. Here, I propose a Coq implementation of a novel structural operational semantics (SOS)-based deductive proof method, as introduced by Ștefan Ciobâcă, Dorel Lucanu and Sebastian Buruiană [3], whose aim is to address program equivalence. First, we take a look at a formal definition of a hybrid, custom programming language named Spaß that exhibits both imperative and functional aspects, then we define its semantics encoded as a Logically Constrained Term Rewriting System (LCTRS), we define the notion of full functional equivalence, and then we construct a set of automation tactics using Coq's tactic language Ltac, which facilitate and eliminate the verbosity of equivalence proofs that would normally exist without the deployment of said tactics by approximately 95%. Lastly, we take a look at four examples of program pairs for which we prove forward simulation, three of them being focused on an imperative style, two of which employing a loop transformation technique known as loop unrolling, and the last pair being formed of two functional programs, one which utilizes standard recursion, and the other, a more efficient tail recursion method.

Contents

Chapter 1

Introduction

Building correct and reliable systems, both from a hardware and software perspective, is difficult. Very difficult.

The complexity of a system, both intrinsic and extrinsic, accidental and non-accidental, is one of the main factors that is going to dictate not only the engineering costs prior to obtaining our end result, but also the overall quality of our emerging system - i.e, its correctness, safety, security, portability, interoperability, and maintainability, to name a few.

Out of all the properties listed above, however, the first one is the most important - namely, a system's correctness - for if we have a system that possesses all qualities except its own correctness, then what is the point of utilizing it to begin with, if it does not function as it is supposed to, according to a given set of requirements?

Closely tied to correctness is reliability, which is the probability that a given system performs correctly during a specific time or duration - ideally, that probability should converge to 1.

This issue of assuring system correctness and reliability despite the general overarching complexity of modern systems is anything but unknown in Computer Science and Engineering, as well as several other related fields. There have been many proposed solutions to this problem, including high level organizational ones, such as development methodologies, but also ones that are on a lower level, for example, programming

paradigms and design patterns.

In this paper, we will see another solution that is seeing comparatively less use in the software industry at the time of writing, but that is currently adopted by the overwhelming majority of hardware development companies, and that is formal verification, which is a mathematically-based technique that employs formal methods in order to prove (or disprove) the correctness of a given system.

Formal equivalence checking, which falls under the umbrella of formal software and hardware verification, has a wide array of applications, ranging from compiler verification, validation and optimization to cross-version verification and regression testing.

In the sequel, we will explore a Coq approach towards verifying program equivalence, which is based on the work of Ștefan Ciobâcă et al.[3] The adoption of the Coq Proof Assistant for this approach is more than simply a matter of taste, as the nature of Coq's programming language offers us the numerous benefits of functional programming, and Coq's underlying Gallina specification language allows us to define and to prove mathematical statements with relative ease.

A few examples where Coq has been employed within the context of formal verification include:

1. CompCert[4], a fully verified optimizing compiler for C.
2. CertiCrypt[1], a toolset for the construction and verification of cryptographic proofs.
3. CertiKOS[2], an extensible architecture for building certified concurrent OS kernels.

While a major part of this paper may be successfully read and understood on its own, I do recommend reading it alongside an interactive Coq session in either CoqIDE[5] or Proof General[6], for instance, especially in *chapter* 4. In this sense, you may access the entire accompanying Coq project for this paper on GitLab[7]. Furthermore, I also encourage reading

the work of Ștefan Ciobâcă et al.[3] in order to attain an even more comprehensive understanding of this topic.

Chapter 2

Foundations

In this chapter, we will first take a look at some foundational definitions that we will be using throughout the rest of this paper. More specifically, we will examine variable identifiers (section 2.1), $(key, value)$ pair maps (section 2.2), a few well-known fundamentals of category theory (section 2.3), and, finally, we will see a substitution function (section 2.4) that we will be employing for defining the semantics of a language that supports lambda calculus in section 3.2.

2.1 Identifiers

To be able to make use of identifiers, we define a simple inductive type - for our purposes, the letters A through D will suffice as identifier names. To compare two identifiers for equality, we define the function *id_eqb*.

```
Inductive identifier: Type :=
    A | B | C | D.

Definition id_eqb (i i': identifier): bool :=
    match (i, i') with
    | (A, A) => true
    | (B, B) => true
    | (C, C) => true
    | (D, D) => true
    | _      => false
    end.
```

2.2 Maps

Even though we will only require a specific map from identifiers to natural numbers, we may still construct a reusable polymorphic map, where X is the given type of our values:

```
(* A map from identifiers to an arbitrary type X. *)
Definition map (X:Type) := identifier -> X.

(* A specific map type, from identifiers to natural numbers.
 ↪  *)
Definition map_id_nat := map nat.

(* The empty map. Note that our maps are total, which means
 ↪  that every key is mapped to some value. This will not pose
 ↪  any problems in our scenario, as we will define all keys as
 ↪  having a default value, given by d. *)
Definition empty_map {X: Type} (d: X): map X :=
    (fun _ => d).

(* The empty map with all (k, v) pairs being set to (k, 0) by
 ↪  default. *)
Definition map_id_nat_empty := empty_map 0.
```

Along with our map, we will also need two methods for updating it and for finding elements by key. In the case of the update function, we may also define a helpful notation.

```
Definition update {X: Type} (k: identifier) (v: X) (m: map X):
 ↪  map X :=
    fun k' => if id_eqb k k' then v else m k'.
Definition find {X: Type} (k: identifier) (m: map X): X := m k.

Notation "k '|->' v 'IN' m" := (update k v m) (at level 60,
 ↪  right associativity).

Example myMap :=
    A |-> 1 IN
    B |-> 2 IN
    map_id_nat_empty.
```

2.3 Functors, Applicatives and Monads

For our substitution function that we will see in section 2.4, we exert the power of functors and applicatives, as known in category theory, in order to be able to work with the *option* monad that Coq provides. Moreover, even though it is not absolutely required, we will also be using a monad instance for *option*, whose deployment we will see in section 3.2.

Do note that while *option* is already a monad (and thus also an applicative and a functor), Coq does not provide the methods associated to it (i.e, 'bind' and 'return') in an off-the-shelf manner. Hence, by functor, applicative or monad instances, we should think of the actual methods that are required to consider a particular type as being either a functor, an applicative or a monad.

Let us briefly cover the implementation of each of these, as they are not directly standardized in Coq:

```
(* The following simply defines infix function application, a
 ↪  standard that is present in Haskell, denoted by the symbol
 ↪  '$'. While it may seem at first glance that such a
 ↪  definition may be useless, it does actually enable us to
 ↪  avoid excessive parantheses. For instance, instead of 'f (g
 ↪  (h (i x)))', we may instead write 'f $ g $ h $ i x'. *)
Definition funApp {A B: Type} (f: A -> B) (x: A): B := f x.

(* Functor instance of the option monad. *)
Definition fmap {A B} (f: A -> B) (a: option A): option B :=
    match a with
    | Some a' => Some (f a')
    | _       => None
    end.

(* Applicative instance of the option monad. *)
Definition app {A B} (f: option (A -> B)) (a: option A): option
 ↪  B :=
    match f with
    | Some f' => fmap f' a
```

```
    | _       => None
    end.

(* Monad instance of the option monad. *)
Definition bind {A B} (m: option A) (a: A -> option B): option
 ↪  B :=
    match m with
    | Some m' => a m'
    | _       => None
    end.
Definition ret {A} (a:A): option A :=
    Some a.

(* Notations popularized by Haskell for the above. *)
Infix "$"   := funApp (at level 100, right associativity).
Infix "<$>" := fmap (at level 90, left associativity).
Infix "<*>" := app (at level 90, left associativity).
Infix ">>=" := bind (at level 90, left associativity).
```

2.4 Lambda Substitution

For our tackling of operational semantics in section 3.2, we are going to need two auxiliary methods, namely the following:

1. A method that checks whether a given expression is a value (a value being either a natural number, a boolean, an identifier, a lambda abstraction, or a fixpoint).

2. And another one that performs a substitution for reducing lambda applications. In our case, we will see a non-capture avoiding type of substitution, as it is enough for the examples that we will examine in chapter 4.

Beginning with (1), we have:

```
Definition isval (e: exp): bool :=
    match e with
    | ENat _   => true
```

```
  | EBool _  => true
  | EId _    => true
  | EAbs _ _ => true
  | EFix _ _ => true
  | _        => false
  end.
```

As for (2):

```
(* Substitute variable x with expression e' in e. *)
Fixpoint subst (x: identifier) (e' e: exp): option exp :=
    if (isval e') then
        match e with
        | EId x'      =>
            Some $ if (id_eqb x x') then e' else EId x'
        | EApp e1 e2  =>
            EApp <$> subst x e' e1 <*> subst x e' e2
        | EAbs x' e'' =>
            if (id_eqb x x') then Some $ EAbs x' e'' else EAbs
             ↪  x' <$> subst x e' e''
        | EFix x' e'' =>
            if (id_eqb x x') then Some $ EFix x' e'' else EFix
             ↪  x' <$> subst x e' e''
        | ENat i      =>
            Some $ ENat i
        | EBool b     =>
            Some $ EBool b
        | EAdd e1 e2  =>
            EAdd <$> subst x e' e1 <*> subst x e' e2
        | ESub e1 e2  =>
            ESub <$> subst x e' e1 <*> subst x e' e2
        | EMul e1 e2  =>
            EMul <$> subst x e' e1 <*> subst x e' e2
        | EDiv e1 e2  =>
            EDiv <$> subst x e' e1 <*> subst x e' e2
        | EEq e1 e2   =>
            EEq  <$> subst x e' e1 <*> subst x e' e2
        | ELt e1 e2   =>
            ELt  <$> subst x e' e1 <*> subst x e' e2
```

```
    | EAnd e1 e2  =>
      EAnd <$> subst x e' e1 <*> subst x e' e2
    | ENot e''    =>
      ENot <$> subst x e' e''
    | ESkip       =>
      Some ESkip
    | ESeq e1 e2  =>
      ESeq <$> subst x e' e1 <*> subst x e' e2
    | EAss x' e'' =>
      match e' with
      | EId x'' =>
        if (id_eqb x x') then EAss x'' <$> subst x e'
        ↪  e'' else EAss x' <$> subst x e' e''
      | _       =>
        EAss x' <$> subst x e' e''
      end
    | EWhile e1 e2  =>
      EWhile <$> subst x e' e1 <*> subst x e' e2
    | EIte e1 e2 e3 =>
      EIte   <$> subst x e' e1 <*> subst x e' e2 <*>
      ↪  subst x e' e3
    | EHole         =>
      Some EHole
    end
else None.
```

Chapter 3

Approach

In order to be able to specify programs, we are going to require a language. Let us call it Spaß. In this chapter, we will proceed by examining the syntax and semantics of Spaß in section 3.1 and section 3.2 respectively, after which we will define the notion of full functional equivalence in section 3.3. Lastly, we will also take a look at a few tactics in section 3.4 that will aid us in chapter 4, when we tackle equivalence proofs.

3.1 Spaß Syntax

We define Spaß as being the language described by the following notation in Backus-Naur Form (BNF), where:

1. Expressions can be:
   (a) Primitives.
   (b) Arithmetic operations.
   (c) Logical operations.
   (d) Imperative commands.
   (e) Lambda abstractions, fixpoints, or lambda applications.
   (f) A hole/box whose evaluation is pending.
2. A stack can be either:
   (a) The empty stack.

(b) An expression, followed by a stack.

3. Finally, a configuration is simply defined as a double, whose first element is a stack and its second, a map from identifiers to natural numbers.

```
exp ::=
    Nat | Bool | Id                                    |
    exp + exp | exp - exp | exp * exp | exp / exp      |
    exp ==? exp | exp <? exp | exp & exp | !exp        |
    SKIP                                               |
    exp; exp                                           |
    exp ::= exp                                        |
    WHILE exp DO exp END                               |
    ITE exp THEN exp ELSE exp ETI                      |
    λexp.exp                                           |
    μexp.exp                                           |
    exp exp                                            |
    □.

stack ::=
    EmptyStack |
    exp ⇝ stack.

cfg ::= [stack | map_id_nat].
```

We will now proceed by translating the above (more informal) definitions, into more formal ones, using Coq.

```
Inductive exp: Type :=
    (* Primitives. *)
    | ENat  (x:    nat)
    | EBool (x:    bool)
    | EId   (x:    identifier)
    (* Arithmetic operations. *)
    | EAdd  (e e': exp)
    | ESub  (e e': exp)
    | EMul  (e e': exp)
    | EDiv  (e e': exp)
```

```
    (* Logical operations. *)
    | EEq   (e e': exp)
    | ELt   (e e': exp)
    | EAnd  (e e': exp)
    | ENot  (e:    exp)
    (* Lambda calculus. *)
    | EAbs  (x:    identifier) (e: exp)
    | EFix  (x:    identifier) (e: exp)
    | EApp  (e e': exp)
    (* Hole/Box. *)
    | EHole.

Inductive stack: Type :=
    | EmptyStack
    | Stack (e: exp) (s: stack).

Inductive cfg: Type :=
    Cfg (s: stack) (m: map_id_nat).
```

Finally, a few notations for our our *exp*, *stack* and *cfg* types:

```
(* The following coercions will allow Coq to interpret our
 ↪  primitive expression types that we have defined as elements
 ↪  of type exp, whenever it is clear from the context that it
 ↪  ought to do so. *)
Coercion ENat : nat >-> exp.
Coercion EBool: bool >-> exp.
Coercion EId  : identifier >-> exp.

Infix "+"      := EAdd (at level 50, left associativity):
 ↪  SpassScope.
Infix "-"      := ESub (at level 50, left associativity):
 ↪  SpassScope.
Infix "*"      := EMul (at level 40, left associativity):
 ↪  SpassScope.
Infix "/"      := EDiv (at level 40, left associativity):
 ↪  SpassScope.
Infix "==?"    := EEq (at level 55, left associativity):
 ↪  SpassScope.
```

```
Infix "<?"      := ELt (at level 54, left associativity):
 ↪  SpassScope.
Infix "&"       := EAnd (at level 56, left associativity):
 ↪  SpassScope.
Notation "! b" := (ENot b) (at level 30, right associativity):
 ↪  SpassScope.

Notation "'SKIP'" :=
    (ESkip):
    SpassScope.
Notation "c1 ; c2" :=
    (ESeq c1 c2)
    (at level 70, right associativity): SpassScope.
Notation "x '::=' e" :=
    (EAss x e)
    (at level 60): SpassScope.
Notation "'WHILE' b 'DO' c 'END'" :=
    (EWhile b c)
    (at level 70, right associativity): SpassScope.
Notation "'ITE' b 'THEN' c1 'ELSE' c2 'ETI'" :=
    (EIte b c1 c2)
    (at level 70, right associativity): SpassScope.

Notation "e ~> st" := (Stack e st) (at level 80, right
 ↪  associativity).
Notation "[s | m]"  := (Cfg s m) (at level 90).
```

As for the lambda calculus, there are currently no specific notations defined, due to it being a bit more difficult to find proper ones in Coq (especially for the lambda applications), however, for better readability, we will still be using the writing style as described in the BNF notation for our lambda expressions, presented earlier in this chapter.

3.2 LCTRS Encoding of Spaß Semantics

At this point, we're able to start outlining our semantics, encoded as a LCTRS, which is simply a set of logically constrained rewrite rules. For

simplicity, I will not cover the actual implementation in Coq for the operational semantics, as it is a bit lengthy, however, it will suffice for us to see Spaß's semantics defined by a set of structural operational semantics (SOS) rules.

We use a function called 'runStep' to perform one rewrite according to the SOS rules that we will examine shortly.

```
Definition runStep (config: cfg): option cfg.
(* Full definition omitted. *)
```

If we would like to do multiple rewrites in a row, or, in other words, to perform multiple program steps at once, then we may adopt the following function:

```
Fixpoint runNSteps (n: nat) (config: cfg): option cfg :=
    match n with
    | O    => ret config
    | S n' => runStep config >>= fun config' =>
        runNSteps n' config'
    end.
```

For the inference system of our LCTRS, we will need a function that is a bit different than the function *isval* that we've seen in section 2.4 - namely, we require *isval_except_id*, which checks whether a given expression is a value, as we've seen before, but not an *EId* _. On top of that, we will need a *flip* function that simply flips the order of the arguments that it receives. More precisely:

```
Definition isval_except_id (e: exp): bool :=
    match e with
    | ENat _   => true
    | EBool _  => true
    | EAbs _ _ => true
    | EFix _ _ => true
    | _        => false
    end.

Definition flip {A B C: Type} (f: A -> B -> C) (y: B) (x: A): C
↪  := f x y.
```

```
(* Will print 10. *)
Compute flip Nat.sub 10 20.
```

Finally, we define the LCTRS encoding of Spaß's semantics as follows (note that instead of *EHole*, we write $\square$, as seen in section 3.1, for simplicity; similarly, we will also be using the lambda notations in the BNF of our expressions, found in the same section):

Do note that in rule IR_ADD_4, $(n_1 + n_2)$ in our first configuration is an expression, while $(n_1 + n_2)\%nat$ from our second configuration is a natural number.

The rules for the arithmetic operations (other than addition) and for the binary logical operations are similar to IR_ADD_0 - IR_ADD_4 - therefore, we will omit them here.

$$\frac{}{runStep\ \$\ [SKIP \rightsquigarrow s|env] = Some\ \$\ [s|env]}\ IR\_SKIP$$

$$\frac{\neg isval\_except\_id(e)}{runStep\ \$\ [x ::= e \rightsquigarrow s|env] = Some\ \$\ [e \rightsquigarrow x ::= \square \rightsquigarrow s|env]}\ IR\_ASSIG\_0$$

$$\frac{\mathrm{n} \in \mathbb{N}}{runStep\ \$\ [n \rightsquigarrow x ::= \square \rightsquigarrow s|env] = Some\ \$\ [x ::= n \rightsquigarrow s|env]}\ IR\_ASSIG\_1$$

$$\frac{\mathrm{n} \in \mathbb{N}}{runStep\ \$\ [x ::= n \rightsquigarrow s|env] = Some\ \$\ [s \mid x\ \mapsto n\ IN\ env]}\ IR\_ASSIG\_2$$

$$\frac{}{runStep\ \$\ [x \rightsquigarrow s|env] = Some\ \$\ [env\ x \rightsquigarrow s|env]}\ IR\_ID\_LOOKUP$$

$$\frac{\neg isval\_except\_id(e_1)}{runStep\ \$\ [e_1 + e_2 \rightsquigarrow s|env] = Some\ \$\ [e_1 \rightsquigarrow \square + e_2 \rightsquigarrow s|env]}\ IR\_ADD\_0$$

$$\frac{\mathrm{n}_1 \in \mathbb{N}}{runStep\ \$\ [n1 \rightsquigarrow \square + e_2 \rightsquigarrow s|env] = Some\ \$\ [n_1 + e_2 \rightsquigarrow s|env]}\ IR\_ADD\_1$$

$$\frac{\mathrm{n}_1 \in \mathbb{N} \wedge \neg isval\_except\_id(e_2)}{runStep\ \$\ [n_1 + e_2 \rightsquigarrow s|env] = Some\ \$\ [e_2 \rightsquigarrow n_1 + \square \rightsquigarrow s|env]}\ IR\_ADD\_2$$

$$\frac{\mathrm{n}_1, n_2 \in \mathbb{N}}{runStep\ \$\ [n_2 \rightsquigarrow n_1 + \square \rightsquigarrow s|env] = Some\ \$\ [n_1 + n_2 \rightsquigarrow s|env]}\ IR\_ADD\_3$$

$$\frac{\mathrm{n}_1, n_2 \in \mathbb{N}}{runStep\ \$\ [n_1 + n_2 \rightsquigarrow s|env] = Some\ \$\ [(n_1 + n_2)\%nat \rightsquigarrow s|env]}\ IR\_ADD\_4$$

$$\frac{\neg isval\_except\_id(e)}{runStep\ \$\ [!e \rightsquigarrow s|env] = Some\ \$\ [e \rightsquigarrow !\square \rightsquigarrow s|env]}\ IR\_NEG\_0$$

$$\frac{\mathrm{b} \in \mathbb{B}}{runStep\ \$\ [b \rightsquigarrow !\square \rightsquigarrow s|env] = Some\ \$\ [!b \rightsquigarrow s|env]}\ IR\_NEG\_1$$

$$\frac{\mathrm{b} \in \mathbb{B}}{runStep\ \$\ [!b \rightsquigarrow s|env] = Some\ \$\ [\overline{b} \rightsquigarrow s|env]}\ IR\_NEG\_2$$

$$\frac{\neg isval\_except\_id(e_1)}{runStep\ \$\ [ITE\ e_1\ THEN\ e_2\ ELSE\ e_3\ ETI \rightsquigarrow s|env] = Some\ \$\ [e_1 \rightsquigarrow ITE\ \square\ THEN\ e_2\ ELSE\ e_3\ ETI \rightsquigarrow s|env]}\ IR\_ITE\_0$$

$$\frac{\mathrm{b}_1 \in \mathbb{B}}{runStep\ \$\ [b_1 \rightsquigarrow ITE\ \square\ THEN\ e_2\ ELSE\ e_3\ ETI \rightsquigarrow s|env] = Some\ \$\ [ITE\ b_1\ THEN\ e_2\ ELSE\ e_3\ ETI \rightsquigarrow s|env]}\ IR\_ITE\_1$$

$$\frac{}{runStep\ \$\ [ITE\ true\ THEN\ e_2\ ELSE\ e_3\ ETI \rightsquigarrow s|env] = Some\ \$\ [e_2 \rightsquigarrow s|env]}\ IR\_ITE\_2$$

$$\frac{}{runStep\ \$\ [ITE\ false\ THEN\ e_2\ ELSE\ e_3\ ETI \rightsquigarrow s|env] = Some\ \$\ [e_3 \rightsquigarrow s|env]}\ IR\_ITE\_3$$

$$\frac{}{runStep\ \$\ [WHILE\ e_1\ DO\ e_2\ END \rightsquigarrow s|env] = Some\ \$\ [ITE\ e_1\ THEN\ (e_2;\ WHILE\ e_1\ DO\ e_2\ END)\ ELSE\ SKIP\ ETI \rightsquigarrow s|env]}\ IR\_WHILE$$

$$\frac{\neg isval\_except\_id(e_1)}{runStep\ \$\ [e_1; e_2 \rightsquigarrow s|env] = Some\ \$\ [e_1 \rightsquigarrow \square; e_2 \rightsquigarrow s|env]}\ IR\_SEQ\_0$$

$$\frac{isval\_except\_id(e_1)}{runStep\ \$\ [e_1 \rightsquigarrow \square; e_2 \rightsquigarrow s|env] = Some\ \$\ [e_1; e_2 \rightsquigarrow s|env]}\ IR\_SEQ\_1$$

$$\frac{isval\_except\_id(e_1)}{runStep\ \$\ [e_1; e_2 \rightsquigarrow s|env] = Some\ \$\ [e_2 \rightsquigarrow s|env]}\ IR\_SEQ\_2$$

$$\frac{\neg isval\_except\_id(e_1)}{runStep\ \$\ [e_1\ e_2 \rightsquigarrow s|env] = Some\ \$\ [e_1 \rightsquigarrow \square\ e_2 \rightsquigarrow s|env]}\ IR\_LAM\_0$$

$$\frac{isval\_except\_id(e_1)}{runStep\ \$\ [e_1 \rightsquigarrow \square\ e_2 \rightsquigarrow s|env] = Some\ \$\ [e_1\ e_2 \rightsquigarrow s|env]}\ IR\_LAM\_1$$

$$\frac{\neg isval\_except\_id(e')}{runStep\ \$\ [(\lambda x.e)\ e' \rightsquigarrow s|env] = Some\ \$\ [e' \rightsquigarrow (\lambda x.e)\ \square \rightsquigarrow s|env]}\ IR\_LAM\_2$$

$$\frac{\neg isval\_except\_id(e')}{runStep\ \$\ [(\mu x.e)\ e' \rightsquigarrow s|env] = Some\ \$\ [e' \rightsquigarrow (\mu x.e)\ \square \rightsquigarrow s|env]}\ IR\_LAM\_3$$

$$\frac{isval\_except\_id(e')}{runStep\ \$\ [e' \rightsquigarrow (\lambda x.e)\ \square \rightsquigarrow s|env] = Some\ \$\ [(\lambda x.e)\ e' \rightsquigarrow s|env]}\ IR\_LAM\_4$$

$$\frac{isval\_except\_id(e')}{runStep\ \$\ [e' \rightsquigarrow (\mu x.e)\ \square \rightsquigarrow s|env] = Some\ \$\ [(\mu x.e)\ e' \rightsquigarrow s|env]}\ IR\_LAM\_5$$

$$\frac{isval\_except\_id(e')}{runStep\ \$\ [(\lambda x.e)\ e' \rightsquigarrow s|env] = Some\ \$\ flip\ Cfg\ env\ <\$>\ (flip\ Stack\ s\ <\$>\ subst\ x\ e'\ e)}\ IR\_LAM\_6$$

$$\frac{isval\_except\_id(e') \wedge subst\ x'\ e'\ e\ =\ Some\ res}{runStep\ \$\ [(\mu x.\lambda x'.e)\ e' \rightsquigarrow s|env] = flip\ Cfg\ env\ <\$>\ (flip\ Stack\ s\ <\$>\ subst\ x\ (\mu x.\lambda x'.e)\ res)}\ IR\_LAM\_7$$

$$\frac{}{runStep\ \$\ [\mu x.e \rightsquigarrow s|env] = Some\ \$\ flip\ Cfg\ env\ <\$>\ (flip\ Stack\ s\ <\$>\ subst\ x\ (\mu x.e)\ e)}\ IR\_LAM\_8$$

3.3 Functional Equivalence

It is time for us to address the core of our method to establish whether two programs specified using Spaß find themselves in an equivalence relation or not.

To do so, we must define what a complete path is. As a remark, we will not be talking about branching paths, as we most often see with real programming languages, since Spaß only encompasses deterministic programs, and not non-deterministic.

We say that two program configurations P and P'' form a complete path if they are either the same program and *runStep P* evaluates to *None*, **or**, if *runStep P* evaluates to some P' s.t. P' and P'' form a complete path.

In Coq, it is best to define the above as an inductive relation, as it will allow us to employ a larger spectrum of Coq's proof tactics.

```
(* Intuitively, an initial program configuration forms a
 ↪  complete path with the end configuration of the program.
 ↪  *)
Inductive isCompletePath_R: cfg -> cfg -> Prop :=
    | CPR_BaseCase: ∀ P,
        runStep P = None  ⟹  isCompletePath_R P P
    | CPR_IndCase: ∀ P P' P'',
        runStep P = Some P' ∧ isCompletePath_R P' P''  ⟹
         ↪  isCompletePath_R P P''.
```

Evidently, however, it is not enough for us to know that two initial, distinct program configurations form two complete paths to ascertain whether said programs are equivalent or not. For that, we must also analyze their outputs. In our case, a program's output is the set of $(key, value)$ pairs found in its end environment.

Thus, we must now establish **when** we consider two programs to be equivalent. Is it when their end environments are equal? Or when the values of one variable from each environment are equal? Or perhaps some other case?

To be able to formally specify that, we make use of what is known as a base case set[3]. A base case set contains all program configuration doubles that are assumed to be equivalent.

Let us see some examples to understand this concept better:

```
(* A base case set that contains all configuration pairs whose
 ↪  values for variable A are equal. *)
Definition baseCaseSet_varsAEqual: cfg -> cfg -> Prop :=
    fun c1 c2 =>
        match (c1, c2) with
        | ([_ | e1], [_ | e2]) => find A e1 = find A e2
        end.

(* Base case set that contains all configuration pairs whose
 ↪  environments are equal. *)
Definition baseCaseSet_envsEqual: cfg -> cfg -> Prop :=
    fun c1 c2 => c1 = c2.
```

Now that we understand what a complete path and a base case set are, we are ready to define full functional equivalence. Two programs are called fully functionally equivalent[3] whenever they fully simulate one another.

Furthermore, two programs are known to fully simulate each other when for the initial configurations P and Q of each program, we have that for all complete paths starting from P and ending in some P', there exists a complete path in the second program starting from Q and ending in some

Q′ s.t. P′ and Q′ form a base case set (as discussed a bit earlier, since Spaß is a deterministic language, we only truly have that each possible configuration is paired with just one other end configuration to form a complete path).

In other words:

```
Definition fullSimulation (P Q: cfg) (baseCaseSet: cfg -> cfg
↪  -> Prop): Prop :=
    ∀ P', isCompletePath_R P P' ⟹
        ∃ Q', isCompletePath_R Q Q' ∧ baseCaseSet P' Q'.

Definition fullEquivalence (P Q: cfg) (baseCaseSet: cfg -> cfg
↪  -> Prop): Prop :=
    fullSimulation P Q baseCaseSet ∧ fullSimulation Q P
    ↪  baseCaseSet.
```

3.4 Spaß Tactics

We've just seen the definition for full functional equivalence, and that it is based on bisimulation.

From now on, I will use the terms "equivalence" and "simulation" rather interchangeably when talking about proving program equivalence, as proving equivalence implies proving simulation. When not clear from the context, I will state clearly which one of those two notions I am referring to.

Nevertheless, we are not yet done, as we still require some way to prove that a program is able to transition from one configuration to another, through zero or more steps, to ultimately arrive at an end configuration.

While it is definitely possible to write equivalence proofs without any additional aid than what we've seen thus far, one will quickly find that such raw proofs tend to be extremely repetitive, long, and verbose.

To avoid all of those negative aspects, we turn our attention to custom Coq tactics in this section. These tactics will turn out to revive the proofs

that we will see in chapter 4 and eliminate the verbosity that would normally be exhibited without them.

We start with some useful lemmas for stepping left and/or right within our program pair that we would like to check for simulation. We omit their corresponding proof definitions here, as they are not incredibly important for what we are about to see next.

```
Lemma stepLeft: ∀ P Q P' baseCaseSet,
    runStep P = Some P' ∧ fullSimulation P' Q baseCaseSet ⟹
        fullSimulation P Q baseCaseSet.
Proof.
(* Omitted. *)

Lemma stepRight: ∀ P Q Q' baseCaseSet,
    runStep Q = Some Q' ∧ fullSimulation P Q' baseCaseSet ⟹
        fullSimulation P Q baseCaseSet.
Proof.
(* Omitted. *)

Lemma stepLeftN (n:nat): ∀ P Q P' baseCaseSet,
    runNSteps n P = Some P' ∧ fullSimulation P' Q baseCaseSet
     ↪ ⟹ fullSimulation P Q baseCaseSet.
Proof.
(* Omitted. *)

Lemma stepRightN (n:nat): ∀ P Q Q' baseCaseSet,
    runNSteps n Q = Some Q' ∧ fullSimulation P Q' baseCaseSet
     ↪ ⟹ fullSimulation P Q baseCaseSet.
Proof.
(* Omitted. *)
```

Using the lemmas above, we may now define our first tactics for performing L/R steps, applying Coq's tactic language, Ltac.

```
(* Perform n left steps. *)
Ltac stepLN n :=
```

```
(* By using `eapply (stepLeftN n)`, we generate a goal
 ↪  `runNSteps n P = Some ?P' ∧ fullSimulation ?P' Q
 ↪  baseCaseSet`, where P and Q are our left and right program
 ↪  configurations respectively. We split this goal, and apply
 ↪  reflexivity on the first conjunct, to get that ?P' is, in
 ↪  fact, the configuration that is reached by performing `n`
 ↪  steps starting from P.

   As for the second conjunct, we just perform some unfolding
 ↪  operations that will simplify our programs' environments.

   After utilizing this tactic, we thus obtain a goal that lets
 ↪  us prove `fullSimulation` between the left program, after
 ↪  having gone through `n` steps, and the untouched right
 ↪  program.
*)
eapply (stepLeftN n); split;
[reflexivity | unfold find, update, id_eqb];
simpl.

(* Perform n right steps. This tactic is quite symmetric with
 ↪  the previous one. *)
Ltac stepRN n :=
    eapply (stepRightN n); split;
    [reflexivity | unfold find, update, id_eqb];
    simpl.

(* Some extra intuitive stepping tactics. *)
Ltac stepLNRM n m := stepLN n; stepRN n.
Ltac stepLRN n    := stepLNRM n n.

Ltac stepL  := stepLN 1.
Ltac stepR  := stepRN 1.
Ltac stepLR := stepLRN 1.

(* `stepLUntilDone` and `stepRUntilDone` will basically keep
 ↪  attempting to step left and right respectively, until that
 ↪  is no longer possible.
```

```
   This will be vital to employ for our upcoming proof
↪  automation tactics. *)
Ltac stepLUntilDone  := repeat stepL.
Ltac stepRUntilDone  := repeat stepR.
Ltac stepLRUntilDone := stepLUntilDone; stepRUntilDone.
```

A common pattern that one will notice after having programmed for some time is that the vast majority of programs out there exhibit branching (not in a non-deterministic sense, but from a computational point of view). Branching is a direct result of conditional statements, such as the standard *if* statement that we have built into Spaß. Hence, upon meeting any branching point, we may no longer continue program execution until we compute the boolean value for the current condition.

This is no different in our scenario. When performing a certain number of steps within our two programs for which we want to prove or disprove simulation, we will, at some point, have to evaluate boolean conditions, if our underlying programs contain *if* or *while* statements, in the case of Spaß.

Thus, to further be able to automate our equivalence proofs, we will need a tactic for destructing these conditions. With Spaß and the type of programs we will see in chapter 4, a tactic for destructing linear inequalities will suffice.

```
(* First, we define a "native" linear inequality destructing
↪  tactic that receives as argument an equation name that will
↪  track the boolean value of our inequality in each case.

   This tactic is not meant to be called on its own, as we want
↪  to create a notation that will allow us to supply no
↪  argument, and let Coq generate an equation name by itself.

   For our purposes, it is enough to handle less-than and
↪  equality conditions. In both cases, we are simply
↪  destructing the condition that we have, obtaining a goal
↪  where we assume the condition evaluates to true, and
↪  another one in which we assume it evaluates to false.
```

```
   Afterwards, to further facilitate automation, we assume that
↪  the negation of each one of those evaluated conditions is
↪  true, and we attempt to arrive at a contradiction - that
↪  way, we may be able to resolve one of the destructed goals
↪  purely through automation. If that is not possible, then we
↪  just continue as normal. *)
Ltac destLinIneq_native eqName :=
    match goal with
    | [ |- context[Nat.ltb ?x ?y] ] =>
        destruct (Nat.ltb x y) eqn:eqName;
        [(* case Nat.ltb x y = true *)
         apply PeanoNat.Nat.ltb_lt in eqName;
         try (
            assert (~ x < y) by lia;
            contradiction
         ) |
         (* case Nat.ltb x y = false *)
         apply PeanoNat.Nat.ltb_ge in eqName;
         try (
            assert (~ y <= x) by lia;
            contradiction
         )]
    | [ |- context[Nat.eqb ?x ?y] ] =>
        destruct (Nat.eqb x y) eqn:eqName;
        [(* case Nat.eqb x y = true *)
         apply PeanoNat.Nat.eqb_eq in eqName;
         try (
            assert (~ x = y) by lia;
            contradiction
         ) |
         (* case Nat.eqb x y = false *)
         apply PeanoNat.Nat.eqb_neq in eqName;
         try (
            assert (~ x <> y) by lia;
            contradiction
         )]
    end.
```

```
(* Now, we may define our notations, as mentioned previously.
 ↪  `fresh` is a method provided by Coq, to generate a new
 ↪  hypothesis name. *)
Tactic Notation "destLinIneq" :=
    let H := fresh in destLinIneq_native H.
Tactic Notation "destLinIneq" simple_intropattern(eqName) :=
    destLinIneq_native eqName.

(* We could also use a tactic for clearing our equation that
 ↪  is produced by `destLinIneq` after applying `destLinIneq`
 ↪  itself, in case we do not need it further in our proof. *)
Ltac destAndClearLinIneq_native eqName :=
    destLinIneq eqName; clear eqName.

Tactic Notation "destAndClearLinIneq" :=
    let H := fresh in destAndClearLinIneq_native H.
Tactic Notation "destAndClearLinIneq"
 ↪  simple_intropattern(eqName) :=
    destAndClearLinIneq_native eqName.
```

With the last two linear inequality destructing tactics in our arsenal, we are now able to come up with yet another useful one - namely, a tactic that performs L/R steps until **after** an *if* statement.

Note that *while* loops are themselves also defined using *if* statements, according to the LCTRS encoding of Spaß's semantics that we have seen in section 3.2 - this means that the following tactic may be deployed in an automated way, in order to step past program loops.

```
Ltac stepLUntilAfterIte :=
    stepLUntilDone;
    repeat destAndClearLinIneq;
    stepL.

Ltac stepRUntilAfterIte :=
    stepRUntilDone;
    repeat destAndClearLinIneq;
    stepR.
```

```
Ltac stepLRUntilAfterIte := stepLUntilAfterIte;
 ↪  stepRUntilAfterIte.
```

We now conclude this section with a slightly more complex tactic that will basically keep stepping left and right, also attempting to step through loops and *if* conditions, until it can no longer do so. Subsequently, we attempt proving full simulation for the two configurations that we have arrived to in the two programs.

Evidently, this tactic will only succeed if there is a "clear" path from the starting configuration pair to the end configuration pair - otherwise, the proof for full simulation will fail, as we will not have obtained a complete path pair.

```
Ltac fullSim base_case_set :=
    repeat (
        stepLRUntilDone;
        repeat destAndClearLinIneq
    );
    (* By unfolding `fullSimulation`, we obtain a new goal
     ↪  that looks like this:

       ∀ P', isCompletePath_R leftConfig P' ⟹ ∃ Q',
 ↪  isCompletePath_R rightConfig Q' ∧ base_case_set P' Q'.

       ...where `leftConfig` and `rightConfig` are our left and
 ↪  right program configurations prior to unfolding. *)
    unfold fullSimulation;
    (* We assume that `∀ P', isCompletePath_R leftConfig P'`
     ↪  holds, after which we apply `eexists` to our goal so
     ↪  that we postpone revealing Q'.

       Then, we split the conjunction that follows. *)
    intros P' H_CPR; (eexists; split;
    [(* To prove `isCompletePath_R rightConfig ?Q'`, we apply
     ↪  inversion to our hypothesis, `H_CPR`, which is
     ↪  `isCompletePath_R leftConfig P'`. *)
     inversion H_CPR as [| config config' config'' contra];
```

```
 [ (* The case where `H_CPR` is destructed to
  ↪  `CPR_BaseCase`. This is the correct case - hence, we
  ↪  are able to prove our goal `isCompletePath_R
  ↪  rightConfig ?Q'` easily. *)
   apply CPR_BaseCase; compute; reflexivity |
   (* This is the impossible case, where `H_CPR` is
    ↪  destructed to `CPR_IndCase`. We know that our left
    ↪  configuration ought to be the end of a complete
    ↪  path, therefore, we are able to apply the principle
    ↪  of explosion. *)
   destruct contra as [contra]; inversion contra
 ] |
 (* Similarly, we prove that P' and Q' are within our base
  ↪  case set by applying inversion on `H_CPR`. *)
 unfold base_case_set;
 inversion H_CPR as [| config config' config'' contra];
 [(* Case when `H_CPR` is `CPR_BaseCase`. *)
  unfold find; try (congruence || reflexivity) |
  (* Case when `H_CPR` is `CPR_IndCase`. *)
  destruct contra as [contra]; inversion contra
 ]
]).
```

Chapter 4

Examples

Having closely examined Spaß's syntax, semantics, as well as a set of subsidiary tactics that will help expedite proof automation, we are now prepared to focus our attention on a few examples of programs for which we prove full equivalence.

We will only take a look at the "forward" direction for our proofs, meaning that we will only prove forward full simulation, as the "backward" simulation proof for all of these program pairs is quite analogous.

Moreover, as Ștefan Ciobâcă et al. have shown[3], some program equivalence proofs may also require a set of auxiliary goals (also called circularities) in order to prove the main goal of simulation. We will discover such secondary goals in the sequel by analyzing the execution of each program.

Lastly, for the first three examples (*section* 4.1, *section* 4.2 and *section* 4.3) that we will examine together, we will adopt the base case set definition that we have seen in *section* 3.3 - namely, *baseCaseSet_varsAEqual*, which contains the program configuration doubles whose environment values for A are equal. The reason we consider the program pairs found in these three examples to be equivalent based on the equality of their A variables is because the variable A will store the value of the sum that is computed in each case.

In the fourth example (*section* 4.4), we will be using *baseCaseSet_envsEqual* that we have also defined in *section* 3.3. This particular set is formed of the

configuration doubles whose **environments** are equal.

4.1 $\sum_1^{n-1}$ and $\sum_0^{n-1}$

First, we begin with two simple imperative programs. Let us denote them as *prog_sum1ToPredN* and *prog_sum0ToPredN* respectively. As their names suggest, the first one of these programs simply calculates the sum between 1 and (n-1), and the second one, the sum between 0 and (n-1).

Their Coq definitions are as follows:

```
Example prog_sum1ToPredN (n:nat) :=
    [A ::= 0; B ::= 1; C ::= n;
     WHILE (B <? C) DO
        A ::= A + B; B ::= B + 1
     END
     ⇝ EmptyStack | map_id_nat_empty].

Example prog_sum0ToPredN (n:nat) :=
    [A ::= 0; B ::= 0; C ::= n;
     WHILE (B <? C) DO
        A ::= A + B; B ::= B + 1
     END
     ⇝ EmptyStack | map_id_nat_empty].
```

Should one attempt to prove full simulation for the above program pair without any additional knowledge, however, one will find that it is rather difficult to do so, if not impossible. Hence, as mentioned earlier, we will now try to find an auxiliary goal that will facilitate our proof for full simulation.

For that, let us analyze the configuration of each of our two programs when given the input $n = 5$ to see how they evolve step-by-step. For simplicity, we will examine only the most important configurations that will allow us to spot an auxiliary goal.

Hence, let $\rightarrow^+$ denote the relation between one program configuration and another future configuration, after having performed one or more

running steps and let ... denote the rest of the stack/environment of the program configuration whenever it is clear from the context what the full stack/environment is.

prog_sum1ToPredN(5) =

```
[A ::= 0; B ::= 1; C ::= 5;
 WHILE (B <? C) DO
    A ::= A + B; B ::= B + 1
 END
 ⇝ EmptyStack | map_id_nat_empty]
→+ [WHILE 1 <? 5 ...                  | C ↦ 5 IN B ↦ 1 IN A ↦ 0 IN map_id_nat_empty]
→+ [A ::= A + B; B ::= B + 1; ...     | C ↦ 5 IN B ↦ 1 IN A ↦ 0 IN map_id_nat_empty]
→+ [WHILE 2 <? 5 ...                  | B ↦ 2 IN A ↦ 1 IN ...]
→+ [A ::= A + B; B ::= B + 1; ...     | B ↦ 2 IN A ↦ 1 IN ...]
→+ [A ::= A + B; B ::= B + 1; ...     | B ↦ 3 IN A ↦ 3 IN ...]
→+ [A ::= A + B; B ::= B + 1; ...     | B ↦ 4 IN A ↦ 6 IN ...]
→+ [WHILE 5 <? 5 DO ...               | B ↦ 5 IN A ↦ 10 IN ...]
→+ [EmptyStack                        | B ↦ 5 IN A ↦ 10 IN ... C ↦ 5 IN ...]
↛+
```

prog_sum0ToPredN(5) =

```
[A ::= 0; B ::= 0; C ::= 5;
 WHILE (B <? C) DO
    A ::= A + B; B ::= B + 1
 END
 ⇝ EmptyStack | map_id_nat_empty]
→+ [WHILE 0 <? 5 ...                  | C ↦ 5 IN B ↦ 0 IN A ↦ 0 IN map_id_nat_empty]
→+ [A ::= A + B; B ::= B + 1; ...     | C ↦ 5 IN B ↦ 0 IN A ↦ 0 IN map_id_nat_empty]
→+ [WHILE 1 <? 5 ...                  | B ↦ 1 IN A ↦ 0 IN ...]
→+ [A ::= A + B; B ::= B + 1; ...     | B ↦ 1 IN A ↦ 0 IN ...]
→+ [WHILE 2 <? 5 ...                  | B ↦ 2 IN A ↦ 1 IN ...]
→+ [A ::= A + B; B ::= B + 1; ...     | B ↦ 2 IN A ↦ 1 IN ...]
→+ [A ::= A + B; B ::= B + 1; ...     | B ↦ 3 IN A ↦ 3 IN ...]
→+ [A ::= A + B; B ::= B + 1; ...     | B ↦ 4 IN A ↦ 6 IN ...]
→+ [WHILE 5 <? 5 DO ...               | B ↦ 5 IN A ↦ 10 IN ...]
→+ [EmptyStack                        | B ↦ 5 IN A ↦ 10 IN ... C ↦ 5 IN ...]
↛+
```

We observe that the configurations whose B and A variables are highlighted red and cyan in the first and second program respectively in fact find themselves pairwise in a full simulation relation.

Thus, we are first going to prove this fact, and then use this proven auxiliary goal/circularity to demonstrate the full simulation of our programs.

```
Lemma sum1ToPredN_sim_sum0ToPredN_G2:
```

```
    (forall n i e e',
     n >= 2 /\ 1 <= i /\ i <= (n-1)%nat
     /\ e A = e' A
     /\ e B = i /\ e' B = i
     /\ e C = n /\ e' C = n ->
        fullSimulation
            ([(A ::= A + B; B ::= B + 1);
              WHILE (B <? C) DO
                A ::= A + B; B ::= B + 1
              END
            ⇝ EmptyStack | e])
            ([(A ::= A + B; B ::= B + 1);
              WHILE (B <? C) DO
                A ::= A + B; B ::= B + 1
              END
            ⇝ EmptyStack | e'])
            baseCaseSet_varsAEqual).
Proof.
    (* We first introduce our quantified variables and
     ↪  hypotheses. *)
    intros n i e e'
        [H_n_ge_2               [
         H_1_le_i               [
         H_i_le_n               [
         H_e_A_eq_e'_A          [
         H_e_B_eq_S_i           [
         H_e'_B_eq_S_i          [
         H_e_C_eq_n H_e'_C_eq_n
        ]]]]]]].
    (* Then, we remember (n - 1 - i), on which we will be
     ↪  performing induction shortly. *)
    remember (Nat.sub (n-1)%nat i)
        as   sub_nMinus1_i
        eqn: Heq_sub_nMinus1_i.
    (* In order to obtain a strong enough induction hypothesis
     ↪  after proving our base case, we generalize all
     ↪  variables. *)
    generalize dependent n;
```

```
    generalize dependent e;
    generalize dependent e';
    generalize dependent i;
    generalize dependent sub_nMinus1_i.
  (* We perform induction on (n - 1 - i) so that in our
  ↪ base case, (n - 1) = i. This means that we will find
  ↪ ourselves just before the end of our last loop,
  ↪ allowing us to easily prove full simulation from that
  ↪ point. *)
  induction sub_nMinus1_i as [| sub_nMinus1_i
  ↪ IH_sub_nMinus1_i]; intros.

  (* (n - 1 - i), base case. *)
  - (* This case is very straight forward. Applying the
  ↪ automated tactic `fullSim` that we've defined in
  ↪ chapter 3_4, we trivially solve this goal. *)
    assert (H_nMinus1_eq_i: (n-1)%nat=i) by lia.
    fullSim baseCaseSet_varsAEqual.

  (* (n - 1 - i), inductive case. *)
  - (* In our inductive case, we run a few L & R steps until
  ↪ we get to the point where we are able to apply our
  ↪ induction hypothesis. *)
    stepLRUntilDone.
    assert (H_ite_eq: Nat.ltb (e B + 1) (e C)=Nat.ltb (e' B +
    ↪ 1) (e' C)) by congruence; rewrite H_ite_eq.
    repeat destAndClearLinIneq.
    stepLR.
    apply IH_sub_nMinus1_i with
      (i := S i)
      (n := e C);
      repeat split;
      (lia || congruence).
Qed.
```

Now, we may use *sum1ToPredN_sim_sum0ToPredN_G2* for our main goal.

```
Lemma sum1ToPredN_sim_sum0ToPredN_G1:
```

```
    forall n, fullSimulation (prog_sum1ToPredN n)
     ↪  (prog_sum0ToPredN n) baseCaseSet_varsAEqual.
Proof.
    (* We start off by performing induction on our input n. *)
    induction n.
    - (* The base case is when n=0. This can be automatically
     ↪  proven using `fullSim`. *)
      fullSim baseCaseSet_varsAEqual.
    - (* In the inductive case, we run a few steps as long as
     ↪  we are able to. *)
      stepLRUntilDone.
      (* Here, we destruct the inequality (Nat.ltb 1 (S n)),
       ↪  which appears in both programs. *)
      destLinIneq H_ineq_1_Sn.
      + stepLRUntilDone.
        (* Once again, we perform a destruction of (Nat.ltb 2
         ↪  (S n)) this time, and we remember it. *)
        destLinIneq H_ineq_2_Sn.
        * (* At this point, we just perform one more step in
         ↪  each program and then we are able to apply our
         ↪  circularity, G2. The rest of the goals after this
         ↪  are trivially solved by `fullSim`. *)
          stepLR.
          apply sum1ToPredN_sim_sum0ToPredN_G2 with
            (n := S n)
            (i := 2);
            repeat split;
            lia.
        * fullSim baseCaseSet_varsAEqual.
      + fullSim baseCaseSet_varsAEqual.
Qed.
```

4.2 $\sum_1^{n-1}$ and $\sum_1^{n-1}$ (1x Loop Unroll)

In our second example, we will take a look at the program we've seen before in *section* 4.1, namely, *prog_sum1PredN*, which computes the sum between 1 and (n-1), together with the same program, albeit slightly mod-

ified, in order to support one loop unroll.

Loop unrolling is a loop transformation technique that is often used by optimizing compilers, the goal being to boost program efficiency at the expense of increasing the memory required to store extra instructions created by the unroll.

```
Example prog_sum1ToPredN (n:nat) :=
    [A ::= 0; B ::= 1; C ::= n;
     WHILE B <? C DO
         A ::= A + B; B ::= B + 1
     END
     ⇝ EmptyStack | map_id_nat_empty].

Example prog_sum1ToPredN_1LoopUnroll (n:nat) :=
    [A ::= 0; B ::= 1; C ::= n;
     WHILE (B + 1) <? C DO
         A ::= A + B; B ::= B + 1;
         A ::= A + B; B ::= B + 1
     END;
     ITE !(B + 1 <? C) & (B <? C) THEN
         A ::= A + B; B ::= B + 1
     ELSE SKIP ETI
    ⇝ EmptyStack | map_id_nat_empty].
```

As in section 4.1, we now proceed by analyzing the evolution of each program, in order to discover a possible auxiliary goal. We choose $n = 5$ once again, as the configurations that are produced with this input are enough to reveal the pattern that we are looking for.

prog_sum1ToPredN(5) =

```
[A ::= 0; B ::= 1; C ::= 5;
 WHILE (B <? C) DO
    A ::= A + B; B ::= B + 1;
 END
 ⇝ EmptyStack | map_id_nat_empty]
→+ [WHILE 1 <? 5 ...                  | C ↦ 5 IN B ↦ 1 IN A ↦ 0 IN map_id_nat_empty]
→+ [A ::= A + B; B ::= B + 1; ...     | C ↦ 5 IN B ↦ 1 IN A ↦ 0 IN map_id_nat_empty]
→+ [WHILE 2 <? 5 ...                  | B ↦ 2 IN A ↦ 1 IN ...]
→+ [A ::= A + B; B ::= B + 1; ...     | B ↦ 2 IN A ↦ 1 IN ...]
→+ [A ::= A + B; B ::= B + 1; ...     | B ↦ 3 IN A ↦ 3 IN ...]
→+ [A ::= A + B; B ::= B + 1; ...     | B ↦ 4 IN A ↦ 6 IN ...]
```

```
→⁺ [WHILE 5 <? 5 DO ...                    | B ↦ 5 IN A ↦ 10 IN ...]
→⁺ [EmptyStack                             | B ↦ 5 IN A ↦ 10 IN ... C ↦ 5 IN ...]
↛⁺
```

prog_sum1ToPredN_1LoopUnroll(5) =

```
[A ::= 0; B ::= 1; C ::= 5;
 WHILE (B + 1) <? C DO
     A ::= A + B; B ::= B + 1;
     A ::= A + B; B ::= B + 1
 END;
 ITE !(B + 1 <? C) & (B <? C) THEN
     A ::= A + B; B ::= B + 1
 ELSE SKIP ETI
 ⇝ EmptyStack | map_id_nat_empty
→⁺ [WHILE 2 <? 5                           | C ↦ 5 IN B ↦ 1 IN A ↦ 0 IN map_id_nat_empty]
→⁺ [A ::= A + B; B ::= B + 1;
     A ::= A + B; B ::= B + 1; ...         | C ↦ 5 IN B ↦ 1 IN A ↦ 0 IN map_id_nat_empty]
→⁺ [WHILE 4 <? 5 ...                       | B ↦ 1 IN A ↦ 0 IN ...]
→⁺ [A ::= A + B; B ::= B + 1;
     A ::= A + B; B ::= B + 1; ...         | B ↦ 3 IN A ↦ 3 IN ...]
→⁺ [WHILE 6 <? 5 ...                       | B ↦ 5 IN A ↦ 10 IN ...]
→⁺ [ITE !(6 <? 5) & (5 <? 5) ...           | B ↦ 5 IN A ↦ 10 IN ...]
→⁺ [EmptyStack                             | B ↦ 5 IN A ↦ 10 IN ... C ↦ 5 IN ...]
↛⁺
```

Once again, we notice that the configurations marked with red and cyan for the first and second programs respectively find themselves pairwise in a full simulation relation.

We will first prove this and then make use of this fact to support us in our proof for the full simulation between our two programs, as follows:

```
(* Before we proceed to prove G2 and G1, we are first going to
 ↪  require an auxiliary lemma that will allow us to perform
 ↪  induction until (S (S n)), instead of (S n), otherwise,
 ↪  we wouldn't be able to prove G2. *)
Lemma augmented_induction (P: nat -> Prop) :
    P 0 ->
    P 1 ->
    (forall n, P n -> P (S n) -> P (S (S n))) ->
    forall n, P n.
Proof.
    intros H_P0 H_P1 H_ind.
    assert (forall n, P n /\ P (S n)). {
        induction n as [| n [P_n P_Sn]].
```

```
        - split; assumption.
        - split.
          + assumption.
          + apply H_ind; assumption.
    }
    apply H.
Qed.

Lemma sum1ToPredN_sim_sum1ToPredN_1LoopUnroll_G2:
    forall n i e e',
    n >= 3 /\ 1 <= i /\ i <= (n - 2)%nat
    /\ PeanoNat.Nat.Odd i
    /\ e A = e' A
    /\ e B = i /\ e' B = i
    /\ e C = n /\ e' C = n ->
        fullSimulation
            ([(A ::= A + B; B ::= B + 1);
              WHILE (B <? C) DO
                A ::= A + B; B ::= B + 1
              END
             ⇝ EmptyStack | e])
            ([(A ::= A + B; B ::= B + 1;
               A ::= A + B; B ::= B + 1);
              WHILE (B + 1 <? C) DO
                  A ::= A + B; B ::= B + 1;
                  A ::= A + B; B ::= B + 1
              END ⇝
              EHole;
              ITE !(B + 1 <? C) & (B <? C) THEN
                  A ::= A + B; B ::= B + 1
              ELSE SKIP ETI
              ⇝ EmptyStack | e'])
             baseCaseSet_varsAEqual.
Proof.
    (* As with example 1, we first apply intros. *)
    intros n i e e'
        [H_n_ge_3              [
         H_1_le_i              [
```

```
     H_i_le_nMinus2         [
     H_i_odd                [
     H_e_A_eq_e'_A          [
     H_e_B_eq_i             [
     H_e'_B_eq_i            [
     H_e_C_eq_n H_e'_C_eq_n
    ]]]]]]]].
(* This time, we remember (n - 2 - i), as we will be
 ↪  performing augmented induction on that. *)
remember (Nat.sub (Nat.sub n 2) i)
    as   sub_nMinus2_i
    eqn: Heq_sub_nMinus2_i.
(* Variable generalizations to obtain a strong induction
 ↪  hypothesis. *)
generalize dependent n;
    generalize dependent e;
    generalize dependent e';
    generalize dependent i;
    generalize dependent sub_nMinus2_i.
(* Doing induction on (n - 2 - i), we will have that in
 ↪  our first base case, (n - 2) = i holds. Thus, we may
 ↪  trivially prove that goal through automation. *)
induction sub_nMinus2_i
    as [| | sub_nMinus2_i IH_sub_nMinus2_i]
    using augmented_induction; intros.

(* Note that we obtain two base cases, as we are adopting
 ↪  the augmented induction that we've defined earlier.
 ↪  They are both solved trivially by `fullSim`. *)

(* (n - 2 - i), base case 0. *)
- assert (H_nMinus2_eq_i: (n-2)%nat=i) by lia.
  fullSim baseCaseSet_varsAEqual.

(* (n - 2 - i), base case 1. *)
- assert (H_nMinus3_eq_i: (n-3)%nat=i) by lia.
  fullSim baseCaseSet_varsAEqual.
```

```
    (* (n - 2 - i), augmented inductive case. *)
    - (* Here, we step past two loops in our left program, and
     ↪  then past one loop in our right one, since two loop
     ↪  iterations of the first simulates one iteration of the
     ↪  second. *)
      do 2 stepLUntilAfterIte.
      stepRUntilAfterIte.
      (* Now, our induction hypothesis comes to the rescue. *)
      apply IH_sub_nMinus2_i with
          (i := S (S i))
          (n := n);
          try (lia || congruence).
      apply PeanoNat.Nat.Odd_succ_succ.
      assumption.
Qed.

Lemma sum1ToPredN_sim_sum1ToPredN_1LoopUnroll_G1: forall n,
    fullSimulation (prog_sum1ToPredN n)
     ↪  (prog_sum1ToPredN_1LoopUnroll n)
     ↪  baseCaseSet_varsAEqual.
Proof.
    (* As usual, induction on our input n. Our base case is
     ↪  trivial. *)
    induction n.
    - fullSim baseCaseSet_varsAEqual.
    - (* After running a few steps, we are able to apply G2.
     ↪  The remaining goals can be proven by `fullSim`. *)
      stepLRUntilDone.
      destLinIneq H_ineq_1_Sn; destLinIneq H_ineq_2_Sn.
      + stepLR.
          apply sum1ToPredN_sim_sum1ToPredN_1LoopUnroll_G2 with
            (n := S n)
            (i := 1);
            repeat split;
            try (lia || congruence).
          apply PeanoNat.Nat.odd_spec; reflexivity.
      + fullSim baseCaseSet_varsAEqual.
      + fullSim baseCaseSet_varsAEqual.
```

```
Qed.
```

4.3 $\sum_1^{n-1}$ and $\sum_1^{n-1}$ (2x Loop Unroll)

Our third example will be demonstrating the simulation between *prog_sum1ToPredN*, which we have seen in *section* 4.1 and *section* 4.2, and yet another variant of it that now utilizes two loop unrolls instead of one.

```
Example prog_sum1ToPredN (n:nat) :=
    [A ::= 0; B ::= 1; C ::= n;
     WHILE (B <? C) DO
        A ::= A + B; B ::= B + 1
     END
     ⇝ EmptyStack | map_id_nat_empty].

Example prog_sum1ToPredN_2LoopUnrolls (n:nat) :=
    [A ::= 0; B ::= 1; C ::= n;
     WHILE (B + 2) <? C DO
        A ::= A + B; B ::= B + 1;
        A ::= A + B; B ::= B + 1;
        A ::= A + B; B ::= B + 1
     END;
     ITE !(B + 2 <? C) & (B + 1 <? C) THEN
        A ::= A + B; B ::= B + 1;
        A ::= A + B; B ::= B + 1
     ELSE ITE !(B + 2 <? C) & !(B + 1 <? C) & (B <? C) THEN
        A ::= A + B; B ::= B + 1
     ELSE SKIP ETI ETI
     ⇝ EmptyStack | map_id_nat_empty].
```

This time, we will input $n = 7$ for both programs, since we need to perform a few extra iterations than before to be able to observe a pattern in this scenario.

```
prog_sum1ToPredN(7) =
[A ::= 0; B ::= 1; C ::= 7;
 WHILE (B <? C) DO
    A ::= A + B; B ::= B + 1;
 END
 ⇝ EmptyStack | map_id_nat_empty]
```

```
→⁺ [WHILE 1 <? 7 ...                          | C ↦ 7 IN B ↦ 1 IN A ↦ 0 IN map_id_nat_empty]
→⁺ [A ::= A + B; B ::= B + 1; ...             | C ↦ 7 IN B ↦ 1 IN A ↦ 0 IN map_id_nat_empty]
→⁺ [WHILE 2 <? 5 ...                          | B ↦ 2 IN A ↦ 1 IN ...]
→⁺ [A ::= A + B; B ::= B + 1; ...             | B ↦ 2 IN A ↦ 1 IN ...]
→⁺ [A ::= A + B; B ::= B + 1; ...             | B ↦ 3 IN A ↦ 3 IN ...]
→⁺ [A ::= A + B; B ::= B + 1; ...             | B ↦ 4 IN A ↦ 6 IN ...]
→⁺ [A ::= A + B; B ::= B + 1; ...             | B ↦ 5 IN A ↦ 10 IN ...]
→⁺ [A ::= A + B; B ::= B + 1; ...             | B ↦ 6 IN A ↦ 15 IN ...]
→⁺ [WHILE 7 <? 7 DO ...                       | B ↦ 7 IN A ↦ 21 IN ...]
→⁺ [EmptyStack                                | B ↦ 7 IN A ↦ 21 IN ... C ↦ 7 IN ...]
↛⁺
```

prog_sum1ToPredN_2LoopUnroll(7) =

```
[A ::= 0; B ::= 1; C ::= 7;
 WHILE (B + 2) <? C DO
    A ::= A + B; B ::= B + 1;
    A ::= A + B; B ::= B + 1;
    A ::= A + B; B ::= B + 1
 END;
 ITE !(B + 2 <? C) & (B + 1 <? C) THEN
    A ::= A + B; B ::= B + 1;
    A ::= A + B; B ::= B + 1
 ELSE ITE !(B + 2 <? C) & !(B + 1 <? C) & (B <? C) THEN
    A ::= A + B; B ::= B + 1
 ELSE SKIP ETI ETI
 ⇝ EmptyStack | map_id_nat_empty]
→⁺ [WHILE 3 <? 7                              | C ↦ 7 IN B ↦ 1 IN A ↦ 0 IN map_id_nat_empty]
→⁺ [A ::= A + B; B ::= B + 1;
     A ::= A + B; B ::= B + 1;
     A ::= A + B; B ::= B + 1; ...            | C ↦ 7 IN B ↦ 1 IN A ↦ 0 IN map_id_nat_empty]
→⁺ [WHILE 6 <? 7 ...                          | B ↦ 4 IN A ↦ 6 IN ...]
→⁺ [A ::= A + B; B ::= B + 1;
     A ::= A + B; B ::= B + 1;
     A ::= A + B; B ::= B + 1; ...            | B ↦ 4 IN A ↦ 6 IN map_id_nat_empty]
→⁺ [WHILE 9 <? 7 ...                          | B ↦ 7 IN A ↦ 21 IN ...]
→⁺ [ITE !(9 <? 7) & (8 <? 7) ...              | B ↦ 7 IN A ↦ 21 IN ...]
→⁺ [ITE !(9 <? 7) & !(8 <? 7)
     & (7 <? 7) ...                           | B ↦ 7 IN A ↦ 21 IN ...]
→⁺ [EmptyStack                                | B ↦ 7 IN A ↦ 21 IN ... C ↦ 7 IN ...]
↛⁺
```

As in our last two examples, the configurations whose B and A values are highlighted as red and cyan in the first and second program form a full simulation. Putting this specific circularity to use, we will also be able prove our main goal.

```
(* As with our last example in section 4.2 with only one loop
 ↪  unrolling, we once again are in need of an augmented
 ↪  induction technique to prove our circularity G2. This time,
 ↪  we want to perform induction until S (S (S n)). *)
Lemma augmented_induction' (P: nat -> Prop) :
    P 0 ->
    P 1 ->
    P 2 ->
    (forall n, P n -> P (S n) -> P (S (S n)) -> P (S (S (S
     ↪  n)))) ->
    forall n, P n.
Proof.
    intros H_P0 H_P1 H_P2 H_ind.
    assert (forall n, P n /\ P (S n) /\ P (S (S n))). {
        induction n as [| n [P_n [P_Sn P_SSn]]].
        - repeat split; assumption.
        - repeat split.
            + assumption.
            + assumption.
            + apply H_ind; assumption.
    }
    apply H.
Qed.

Lemma sum1ToPredN_sim_sum1ToPredN_2LoopUnrolls_G2:
    forall n i e e',
    n >= 4 /\ 1 <= i /\ i <= (n - 3)%nat
    /\ (exists x, i = (1 + 3*x)%nat)
    /\ e A = e' A
    /\ e B = i /\ e' B = i
    /\ e C = n /\ e' C = n ->
        fullSimulation
            ([(A ::= A + B; B ::= B + 1);
              WHILE (B <? C) DO
                A ::= A + B; B ::= B + 1
              END
            ⇝ EmptyStack | e])
            ([(A ::= A + B; B ::= B + 1;
```

```
                A ::= A + B; B ::= B + 1;
                A ::= A + B; B ::= B + 1);
               WHILE (B + 2 <? C) DO
                 A ::= A + B; B ::= B + 1;
                 A ::= A + B; B ::= B + 1;
                 A ::= A + B; B ::= B + 1
               END ⇝
               EHole;
               ITE !(B + 2 <? C) & (B + 1 <? C) THEN
                 A ::= A + B; B ::= B + 1;
                 A ::= A + B; B ::= B + 1
               ELSE ITE !(B + 2 <? C) & !(B + 1 <? C) & (B <? C)
                ↪  THEN
                 A ::= A + B; B ::= B + 1
               ELSE SKIP ETI ETI
               ⇝ EmptyStack | e'])
             baseCaseSet_varsAEqual.
Proof.
    (* Introducing our variables and hypotheses. *)
    intros n i e e'
        [H_n_ge_4              [
         H_1_le_i               [
         H_i_le_nMinus3         [
         [x H_i_eq_SmultOf3]    [
         H_e_A_eq_e'_A          [
         H_e_B_eq_i             [
         H_e'_B_eq_i            [
         H_e_C_eq_n H_e'_C_eq_n
        ]]]]]]]].
    (* Remembering the equation (n - 3 - i), to be able to
     ↪  perform induction on that. *)
    remember (Nat.sub (Nat.sub n 3) i)
        as   sub_nMinus3_i
        eqn: Heq_sub_nMinus3_i.
    (* Generalizing all variables. *)
    generalize dependent n;
        generalize dependent x;
        generalize dependent e;
```

```
    generalize dependent e';
    generalize dependent i;
    generalize dependent sub_nMinus3_i.
  (* Performing prime augmented induction on (n - 3 - i),
  ↪  that will generate three simple base cases, which can
  ↪  all be proven by `fullSim`, as usual. *)
  induction sub_nMinus3_i
    as [| | | sub_nMinus3_i IH_sub_nMinus3_i]
    using augmented_induction'; intros.

  (* (n - 3), base case 0. *)
  - assert (H_nMinus3_eq_i: (n-3)%nat=i) by lia.
    fullSim baseCaseSet_varsAEqual.

  (* (n - 3), base case 1. *)
  - assert (H_nMinus4_eq_i: (n-4)%nat=i) by lia.
    fullSim baseCaseSet_varsAEqual.

  (* (n - 3), base case 2. *)
  - assert (H_nMinus5_eq_i: (n-5)%nat=i) by lia.
    fullSim baseCaseSet_varsAEqual.

  (* (n - 3), inductive case. *)
  - (* Similarly as with the programs from section 4.2, we
  ↪  realize that three iterations of the first program
  ↪  produce the same result as only one iteration from the
  ↪  second. After stepping through said number of loop
  ↪  iterations, we may apply our induction hypothesis. *)
    do 3 stepLUntilAfterIte.
    stepRUntilAfterIte.
    apply IH_sub_nMinus3_i
      with (i := S (S (S i)))
      (n := n)
      (x := (x+1)%nat);
      (lia || congruence).
Qed.

Lemma sum1ToPredN_sim_sum1ToPredN_2LoopUnrolls_G1: forall n,
```

```
    fullSimulation (prog_sum1ToPredN n)
    ↪ (prog_sum1ToPredN_2LoopUnrolls n)
    ↪ baseCaseSet_varsAEqual.
Proof.
    (* Induction on n, the base case being directly solved by
    ↪ `fullSim`. *)
    induction n.
    - fullSim baseCaseSet_varsAEqual.
    - (* More or less as in example 2, we just run the right
    ↪ amount of steps, destruct the incoming if conditions,
    ↪ and then apply G2. The other cases can be proven
    ↪ through the `fullSim` tactic. *)
      stepLRUntilDone.
      destLinIneq H_ineq_1_Sn; destLinIneq H_ineq_3_Sn.
      + stepLR.
        apply sum1ToPredN_sim_sum1ToPredN_2LoopUnrolls_G2
            with (n := S n)
            (i := 1);
            repeat split;
            try (lia || congruence).
        exists 0; reflexivity.
      + stepLRUntilDone.
        destLinIneq H_ineq_2_Sn; fullSim
        ↪ baseCaseSet_varsAEqual.
      + fullSim baseCaseSet_varsAEqual.
Qed.
```

4.4 Recursive $\sum_0^n$ and Tail Recursive $\sum_0^n$

For our final example, we will examine two programs that are slightly different compared to what we've seen thus far, as they will be using lambda calculus in order to implement both a recursive and a tail recursive method that will output the sum between 0 and n.

These two programs have been used as a main example by Ștefan Ciobâcă et al. in their paper[3]. Therefore, I will not be reiterating their findings and will simply translate said two programs into Coq, as well as prove

their full simulation directly by demonstrating and utilizing their corresponding circularities that Ștefan Ciobâcă et al. have found.

These are the two programs:

```
Example prog_lambdaSum0ToN (n:nat) :=
    [(μD.λA.
        ITE A ==? 0 THEN 0
        ELSE A + (D (A - 1))
        ETI) n
     ⇝ EmptyStack | map_id_nat_empty].

Example prog_tailRecLambdaSum0ToN (n i a:nat) :=
    [(μD.λA.λB.λC.
        ITE !(B <? A) & !(B ==? A) THEN C
        ELSE D A (B+1) (C+B)
        ETI) n i a
     ⇝ EmptyStack | map_id_nat_empty].
```

For these two programs, we obtain two auxiliary goals[3] (G3 and G2) as follows:

```
(* As described by Ștefan Ciobâcă et al, we require a defined
 ↪  symbol called `reduce` that encapsulates the stack filled
 ↪  with the elements (i + EHole), ((i + 1)%nat + EHole),
 ↪  and so on. This will mimic the postponing of evaluation for
 ↪  our expressions in our first program, which is
 ↪  non-tail-recursive. *)
Fixpoint reduceAux (i n fuel:nat): stack :=
    match fuel with
    | O       =>
        match (Nat.compare i n) with
        | Gt => EmptyStack
        | _  => n + EHole ⇝ EmptyStack
        end
    | S fuel' => i + EHole ⇝ reduceAux (S i) n fuel'
    end.

Definition reduce (i n:nat): stack :=
```

```
    reduceAux i n (Nat.sub n i).

(* Unlike in the last three examples we've seen, here we
↪  actually require two circularities to be able to prove our
↪  main goal, instead of just one. This hints to the
↪  possibility that the more complex two programs are, the
↪  more auxiliary goals one needs to be able to prove their
↪  full simulation or equivalence. *)
Lemma lambdaSum0ToN_sim_tailRecLambdaSum0ToN_G3: forall (n i
↪  a:nat) e,
    1 <= i /\ i <= n ->
        fullSimulation
            ([a ⇝ reduce i n | e])
            ([ITE (negb (Nat.ltb i n) && negb (Nat.eqb i n))
             ↪  THEN a
              ELSE (μD.λA.λB.λC.
                        ITE !(B <? A) & !(B ==? A) THEN C
                        ELSE D A (B+1) (C+B)
                        ETI) n (i+1) (a+i)
             ⇝ EmptyStack | e])
            baseCaseSet_envsEqual.
Proof.
    intros n i a e [H_1_le_i H_i_le_n].
    remember (Nat.sub n i)
        as   sub_n_i
        eqn: Heq_sub_n_i.
    generalize dependent n;
        generalize dependent i;
        generalize dependent a;
        generalize dependent e;
        generalize dependent sub_n_i.
    induction sub_n_i as [| sub_n_i IH_sub_n_i];
    intros; unfold reduce, reduceAux.
    - assert (H_n_eq_i: n=i) by lia.
      rewrite <- H_n_eq_i,
                PeanoNat.Nat.sub_diag,
                PeanoNat.Nat.compare_refl.
      fullSim baseCaseSet_envsEqual.
```

```
      rewrite PeanoNat.Nat.add_comm; reflexivity.
    - assert (H_i_lt_n: i<n) by lia.
      assert (Heq_sub_n_i': sub_n_i = (n - S i)%nat) by lia.
      rewrite <- Heq_sub_n_i,
              Heq_sub_n_i'.
      repeat destAndClearLinIneq H_ineq.
      stepLRUntilDone.
      rewrite PeanoNat.Nat.add_1_r, PeanoNat.Nat.add_comm.
      apply IH_sub_n_i; lia.
Qed.

Lemma lambdaSum0ToN_sim_tailRecLambdaSum0ToN_G2: forall n i,
    i <= (n-1)%nat /\ n>=1 ->
      fullSimulation
          ([(μD.λA.
              ITE A ==? 0 THEN 0
              ELSE A + (D (A - 1))
              ETI)  i
           ⇝ reduce (i+1)%nat n | getEnvOfCfg
            ↪  (prog_tailRecLambdaSum0ToN n 0 0)])
          (prog_tailRecLambdaSum0ToN n 0 0)
          baseCaseSet_envsEqual.
Proof.
    intros n i [H_i_le_predN H_n_ge_1].
    generalize dependent i.
    induction i as [| i IH_i]; intros.
    - stepLRUntilDone.
      repeat destAndClearLinIneq.
      stepLRUntilDone.
      apply lambdaSum0ToN_sim_tailRecLambdaSum0ToN_G3; lia.
    - stepLN 9.
      assert (H_reduce: ((S i + EHole) ⇝ reduce (S (i+1))
       ↪  n)=reduce (i+1)%nat n). {
          unfold reduce, reduceAux.
          assert ((n-(i+1))%nat=S (n - S (S i))%nat) by lia.
          rewrite H, PeanoNat.Nat.add_1_r; reflexivity.
      }
      rewrite H_reduce,
```

```
            PeanoNat.Nat.sub_0_r.
      apply IH_i; lia.
Qed.

Lemma lambdaSum0ToN_sim_tailRecLambdaSum0ToN_G1: forall n,
    fullSimulation (prog_lambdaSum0ToN n)
     ↪  (prog_tailRecLambdaSum0ToN n 0 0)
     ↪  baseCaseSet_envsEqual.
Proof.
    induction n as [| n' IH_n] eqn:H_n.
    - fullSim baseCaseSet_envsEqual.
    - stepLN 6.
      assert (H_reduce: (S n' + EHole ⇝ EmptyStack)=reduce (S
       ↪  n') (S n')). {
          unfold reduce, reduceAux.
          rewrite PeanoNat.Nat.sub_diag,
          PeanoNat.Nat.compare_refl;
          reflexivity.
      }
      rewrite H_reduce.
      stepLN 3.
      assert (H_Sn'_eq_n'Plus1: S n'=(n'+1)%nat) by lia.
      rewrite PeanoNat.Nat.sub_0_r,
              H_Sn'_eq_n'Plus1.
      apply lambdaSum0ToN_sim_tailRecLambdaSum0ToN_G2; repeat
       ↪  split; lia.
Qed.
```

While it may initially seem that this last example was slightly more difficult to prove, that is only because there were a few more points in the proof where certain rewriting rules had to be applied manually. This can be alleviated even further for proofs which involve similar programs by expanding our automation tactics.

Chapter 5

Conclusion

In this paper, we've seen a method to tackle program equivalence proofs using the Coq Theorem Prover by defining our own custom programming language called Spaß, which features both imperative and functional aspects, after which we've defined Spaß's semantics using an LCTRS encoding[3], we've defined full functional equivalence[3], a set of automation tactics for proving said equivalence, and, finally, we've examined four examples of program pairs for which we've proven forward simulation.

Bibliography

[1] *CertiCrypt*. URL: http://certicrypt.gforge.inria.fr.

[2] *CertiKOS*. URL: http://flint.cs.yale.edu/certikos/people.html.

[3] Ștefan Ciobâcă, Dorel Lucanu, and Sebastian Buruiană. "Operationally-based Program Equivalence Proofs using LCTRSs". In: (). URL: https://arxiv.org/abs/2001.09649.

[4] *CompCert*. URL: http://compcert.inria.fr.

[5] *CoqIDE*. URL: https://coq.inria.fr.

[6] *Proof General*. URL: https://proofgeneral.github.io.

[7] Paul Reftu. *(GitLab source code) A Coq Approach Towards SOS-Based Program Equivalence Proofs Using LCTRSs*. URL: http://fmse.info.uaic.ro:8088/paul.reftu/Coq-Program-Equivalence-Using-LCTRSs.