Saarland University
Faculty of Natural Sciences and Technology I

Bachelor's Thesis

Regularity and Linearization of Tail-Recursive Programs

Author:
Clara Schneidewind

Advisor:
Prof. Dr. Gert Smolka

Reviewers:
Prof. Dr. Gert Smolka
Prof. Dr. Sebastian Hack

Submitted: 21st October 2015

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, 21st October, 2015

Abstract

In this thesis we study control flow transformations for abstract imperative programs with actions (e.g. system calls). We take the view that the execution of a program generates a trace of actions.

We employ an equivalence based on total and partial traces. This notion of equivalence leads to a meaningful description of terminating as well as non-terminating programs.

We base our studies on context-free programs - a generalization of regular programs with recursion. Our studies focus on context-free programs with tail recursion. We show and verify equivalence preserving transformations from tail-recursive programs to regular programs and from regular programs to linear tail-recursive programs. In addition, we prove the correctness of a direct transformation from tail-recursive programs to linear tail-recursive programs. Finally, we show how abstract imperative programs with conditionals and loops (abstract IMP) can be encoded using context-free programs. We prove that the equivalence of abstract IMP programs with respect to the big-step semantics is implied by the equivalence of their context-free encodings.

The entire development is carried out in the proof assistant Coq.

Acknowledgements

I want to thank my advisor Prof. Smolka for the numerous meetings and the valuable assistance. I am grateful for his inspiring ideas and for arousing my interest and joy in scientific working.

A special thanks goes to Sebastian and Yannick for proofreading the thesis and to the students of the chair for the interesting discussions that gave valuable impulses for the thesis. I also wish to thank Sigurd for sharing his expertise in writing skills in the field of compiler construction.

I also place on record, my sense of gratitude to Fabian and Yannick, who were always eager to support me in solving Coq, Emacs and LaTeX issues.

Finally, I want to thank my family and Sebastian for patiently listening to me lamenting when proving and writing and for providing the necessary amount of care and chocolate.

Contents

Chapter 1

Introduction

Full sequential composition is a control flow statement used in most high-level imperative programming languages. Compilers translate programs of high-level programming languages to sequences of instructions of register transfer languages. Consequently, compilation needs to dissolve full sequential composition. We call the translation step that dissolves full sequential composition *linearization*.

We motivate linearization of imperative programs with the following program calculating the absolute value of its input. Both programs displayed here show the same input-output behaviour.

```
IN x ;                          IN x ;
if x < 0                        if x < 0
  then x := −x                    then x := −x ; OUT x
  else skip                       else OUT x
end ;                           end
OUT x
```

In the left program, both branches of the the conditional have to execute the same continuation (**OUT** x) after executing the conditional. The program needs to remember this continuation during execution. In the second program all commands are executed consecutively. As both branches of the conditional already contain the continuation, choosing a branch of the conditional directly yields the complete program that remains to be executed. We call such programs that directly reduce to their follow-up program *linear*.

In this thesis, we want to study the correctness of linearization. We aim at realizing this study in a formal mathematical way. To do so, we use an abstraction of imperative programs and formalize linearization in this abstract setting. We carry out the formalization and the correctness proofs with the proof assistant Coq [12].

Context-free programs

The abstract programs we chose to represent imperative programs are *context-free programs*. We obtain context-free programs with the following grammar:

$$s, t \ ::= \ 1 \mid a \mid st \mid s + t \mid \mu x.s \mid x$$

where a ranges over *actions* and x ranges over *variables*. The abstract execution of context-free programs is given as follows:

- The execution of 1 (*skip*) has no effect.
- The execution of an action a invokes this action on the current memory state.
- The execution of a *sequential composition* st consists of the execution of s followed by the execution of t.
- The execution of a *non-deterministic choice* $s + t$ consists of either the execution of s or the execution of t.
- The execution of a *recursion* $\mu x.s$ consists of the execution of the program obtained from s by replacing all free occurrences of the variable x by $\mu x.s$.

We assume that only closed programs are executed. Thus, we do not consider the execution of variables.

Trace semantics

We define a semantics for context-free programs that abstracts from the memory state. Instead of tracking the changes in the memory state, we record the sequences of actions invoked on each possible execution path. We refer to these action sequences as *traces*.

Formally, we characterize a program by the set of its partial and total traces. A partial trace records a partial abstract execution of the program and a total trace records a complete abstract execution of the program. We have partial traces so that we can reason about non-terminating (e.g., reactive) programs.

We call the set of traces described by a program also the *trace language* of the program. Two programs are called equivalent if their trace languages coincide.

Tail-recursive programs

We will study tail-recursive context-free programs. A program is called *tail-recursive* if all variables are in *tail-position*. Examples for tail-recursive programs are:

$$1 \qquad a \qquad \mu x.x \qquad (\mu x.x)a \qquad \mu x.a(bx + c)$$

Tail-recursive programs can be transformed into two kinds of normal forms, namely *regular programs* and *linear programs*. We will give equivalence transformations to these normal forms and show them correct.

Regular programs

Regular programs are used in literature as an abstraction for imperative programs with loops [5]. Regular programs do not contain a general recursion operator but restrict recursions to one of the following forms:

$$\begin{aligned} &\varnothing := \mu x.x \\ &s^* := \mu x.1 + sx && x \textit{ not free in } s \end{aligned}$$

We call s^* *iteration* and $\varnothing$ the *null program*. The syntax of regular programs is given as follows:

$$s, t \ ::= \ 1 \mid a \mid st \mid s + t \mid s^* \mid \varnothing$$

Syntactically, regular programs coincide with regular expressions. Semantically, however, regular expressions and regular programs differ. Using the notion of traces, regular expressions describe a set of total traces while our trace semantics works with partial and total traces.

Nevertheless, the correspondence of regular programs and regular expressions allows us to argue that not every context-free program is regularizable. The total traces of the program

$$\mu x.1 + axb$$

describe the language $a^n b^n$ which is known to be context-free and non-regular. As the total traces of regular programs describe regular languages, there is no equivalent regular program for $\mu x.1 + axb$.

By definition, all regular programs are tail-recursive. Variables can only occur in iterations and the null program where they are located in tail position.

By giving an equivalence transformation from tail-recursive programs to regular programs we conclude that tail-recursive and regular programs describe the same class of trace languages.

Linear programs

We characterize *linear programs* as a subset of context-free programs by restricting sequential compositions to the following form:

$$a; t$$

By definition, every linear program is tail-recursive.

We will verify two kinds of equivalence transformations to establish linearity:

- A transformation from regular programs to linear programs.
- A direct transformation from tail-recursive programs to linear programs.

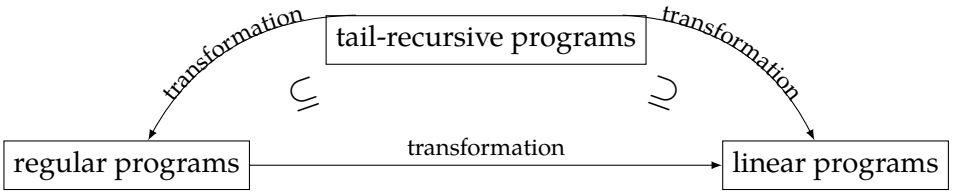

Figure 1.1: Relationship between tail-recursive, linear, and regular programs

Together with the correspondence of regular programs and tail-recursive programs, we conclude that tail-recursive programs, regular-programs, and linear programs describe the same trace languages.

Context-free programs and abstract IMP

To motivate the relationship between context-free programs and imperative programming languages we consider the abstract idealized language IMP [13].

$$c, d \ ::= \ \mathsf{skip} \mid a \mid c;d \mid \mathsf{if}\ b\ c\ d \mid \mathsf{while}\ b\ c$$

In contrast to Winskel's [13] formalization of IMP, we assume abstract actions a instead of variable assignments. In order to give a big-step semantics for IMP [13], we interpret actions as functions changing the memory state. Accordingly, we consider tests b as a subset of actions closed under negation. We interpret tests as partial identities on the memory state which are only defined for conditions evaluating to true in the current memory state.

The following encodings allow us to express IMP programs as context-free programs:

$$\mathsf{if}\ b\ c\ d \rightsquigarrow bc + \overline{b}d$$
$$\mathsf{while}\ b\ c \rightsquigarrow \mu x.\overline{b} + bcx$$

We call the encoding of an IMP program c as a context-free program the *context-free abstraction* of c.

We show the following correspondence of IMP programs and context-free programs:

If the context-free abstractions of two IMP programs c and d are trace equivalent, then c and d are equivalent with respect to the big-step semantics.

Due to this correspondence, we assume that linearization and the proof techniques used for for verification in this thesis carry over to abstract IMP.

As context-free programs can model non-determinism, we conjecture that we can obtain similar results for reactive IMP where divergence matters.

Contribution

We obtained the following results:

- Program equivalence based on traces is a substitutive congruence.
- A verified compiler from tail-recursive programs to regular programs.
- A verified compiler from regular programs to linear tail-recursive programs.
- A verified compiler from tail-recursive programs to linear tail-recursive programs.

Related Work

This thesis is motivated by the translation of the idealized imperative language IMP [13] to the linear intermediate language IL [11]. IL is an intermediate language designed for compiler verification and can be interpreted in an imperative as well as in a functional way. In this thesis, we are only interested in the imperative interpretation of IL.

Fischer and Ladner [5] use regular programs in the context of Propositional Dynamic Logic (PDL). In contrast to the regular programs we consider, Fisher and Ladner additionally assume tests that conditionally block during execution. Together with non-deterministic choices, the regular programs described by Fischer and Ladner give an explanation for Dijkstra's guarded commands [4].

A further generalization of PDL are Kleene algebras with tests (KAT). KAT combines Kleene Algebra with a boolean algebra for tests and gives us a system for equational reasoning on regular programs.

In contrast to the considerations in this thesis, PDL and KAT take a state-based semantics of regular programs as a basis. The trace-based semantics we consider in this thesis dispenses us from requiring tests that follow the laws of boolean algebra, and allows us to use uninterpreted actions instead. Additionally, the trace semantics gives us an informative characterization of diverging programs.

As we use a formalization of programs that replaces the Kleene star with a recursion operator, the results for Kleene algebras (as described by Kozen [6]) are not directly applicable. Nevertheless, the trace semantics fulfils most of the axioms of Kleene algebras not involving Kleene star.

Winter et al. [14] consider context-free expressions with an underlying language semantics and show them to describe context-free languages.

Overview

- Chapter 2. We introduce context-free programs and the trace semantics for context-free programs.
- Chapter 3. We study program equivalence based on traces and prove that program equivalence is a substitutive congruence.
- Chapter 4. We give a verified translation of tail-recursive programs to regular programs and show that tail-recursive programs and regular programs describe the same class of trace languages.
- Chapter 5. We give a compiler from regular programs to linear tail-recursive programs and a compiler from tail-recursive programs to linear tail-recursive programs and prove them correct.
- Chapter 6. We show that equivalence of the context-free abstractions of two abstract IMP programs is a sufficient condition for their equivalence with respect to big-step semantics.

The complete development is carried out with the proof assistant Coq [12].

Chapter 2

Context-Free Programs

We call the programs we will use as a model for imperative programs *context-free programs*. Context-free programs have the same syntax as the context-free expressions described by Winter et al. [14], but will be given a trace semantics instead of a language semantics. The trace semantics allows us to obtain a meaningful description of non-terminating programs as we do not only observe traces of terminating program executions, but also traces resulting from partial executions.

2.1 Syntax

The abstract syntax of context-free programs is defined as follows:

$$s, t, u \ ::= \ 1 \mid x \mid st \mid s + t \mid \mu x.s \qquad (x \in \mathbb{N})$$

Programs of the form x where $x \in \mathbb{N}$ are called **variables**. Variables can occur as *bound variables* if they are introduced by the binder μ or as *free variables*. We write $\boldsymbol{\mathcal{V}s}$ for the set of free variables of a program s.

For technical convenience we accommodate actions as free variables.

We refer to programs of the form $\boldsymbol{st}$ as **sequential compositions** and to those of the form $\boldsymbol{s + t}$ as **choices**. Programs of the form $\boldsymbol{\mu x.s}$ are called **recursions**.

We give an intuitive characterization of program execution.

- The execution of 1 has no effect.
- The execution of x invokes action x.
- The execution of st consists of the execution of s followed by the execution of t.
- The execution of $s + t$ consists of the execution of either s or t.
- The execution of $\mu x.s$ consists of the execution of $s^x_{\mu x.s}$, where $s^x_{\mu x.s}$ is the program obtained from s by replacing all occurrences of x with $\mu x.s$. We call $s^x_{\mu x.s}$ the **unfolding** of $\mu x.s$.

2.2 Semantics

Formally, the execution of a program s yields the sequence of actions invoked during the execution of s. From now on, we call such sequences *traces*. Traces resulting

from a total program execution are called *total traces*, whereas traces obtained from a partial program execution are called *partial traces*.

Traces are formally defined as strings of variables enriched with an ending that indicates whether the trace is partial or total.

$$\xi, \eta, \zeta := \epsilon \mid \# \mid x\xi \qquad (x \in \mathbb{N})$$

Traces ending with $\#$ are called **total** whereas traces ending with ϵ are called **partial**. We call $\#$ and ϵ the **end markers** of a trace and actions occurring in the trace the **elements** of a trace. We denote the variables of a trace ξ as $\mathcal{V}\xi$.

We define a **concatenation operator** for traces.

$$\begin{aligned} \epsilon\,\eta &:= \epsilon \\ \#\,\eta &:= \eta \\ (x\xi)\,\eta &:= x(\xi\,\eta) \end{aligned}$$

Partial traces are absorbing in the sense that concatenation for partial traces yields the identity.

Fact 2.1 (Properties of trace concatenation)

1. If ξ is partial, then $\xi\eta = \xi$.
2. $\xi\# = \xi$
3. $(\xi\eta)\zeta = \xi(\eta\zeta)$
4. $\xi\epsilon$ is partial.

Fact 2.2 (Free variables of concatenated traces)

1. If ξ_1 is partial, then $\mathcal{V}(\xi_1\xi_2) = \mathcal{V}\xi_1$.
2. If ξ_1 is total, then $\mathcal{V}(\xi_1\xi_2) = \mathcal{V}\xi_1 \cup \mathcal{V}\xi_2$.

We give a formal semantics of program execution in terms of an inductive trace predicate.

$$\frac{}{\epsilon/s} \qquad \frac{}{\#/1} \qquad \frac{\xi = \epsilon \vee \xi = \#}{x\xi/x} \qquad \frac{\xi_1/s \qquad \xi_2/t}{\xi_1\,\xi_2/st} \qquad \frac{\xi/s}{\xi/s+t} \qquad \frac{\xi/t}{\xi/s+t} \qquad \frac{\xi/s^x_{\mu x.s}}{\xi/\mu x.s}$$

The rule for sequential composition exploits the properties of the concatenation operator. An incomplete execution of the program s is also an incomplete execution of the the sequential composition st. If s produces a partial trace ξ, this trace can be seen as the composition of ξ with any other trace.

Fact 2.3 If $\xi\eta/s$ and ξ is partial then ξ/s.

Fact 2.4 If ξ/s and ξ is partial, then ξ/st.

The set of all traces of a program s is described by the trace predicate $\boldsymbol{\mathcal{T}}s := \lambda\xi.\,\xi/s$. We call the set of total traces the **language** of s and write $\boldsymbol{\mathcal{L}s}$. We use the notation $\boldsymbol{\xi /\!/\!/ s}$ to say that ξ is a total trace of s and $\boldsymbol{\xi /\!/ s}$ to denote that ξ is a partial trace of s. We say that a program s **diverges** if it has no total traces ($\mathcal{L}s = \emptyset$). A program **silently diverges** if ϵ is its only trace ($\mathcal{T}s = \{\epsilon\}$). The most simple silently diverging program is $\mu x.x$. We call $\mu x.x$ **null program**.

Definition 2.5

$$\varnothing := \mu x.x \qquad \textit{(null program)}$$

Fact 2.6 $\mathcal{T}\varnothing = \{\epsilon\}$

Note that every program has at least one trace.

Fact 2.7 $\epsilon \in \mathcal{T}s$

A trace contains only free variables of a program.

Fact 2.8 If ξ/s, then $\mathcal{V}\xi \subseteq \mathcal{V}s$

The traces of a program are closed under partial prefixes.

Fact 2.9 If ξ/s, then $\xi\epsilon/s$.

Fact 2.10 (Prefix Closedness) If $x_1 \ldots x_n\xi/s$, then $x_1 \ldots x_n\epsilon/s$.

Chapter 3

Program Equivalence

We consider two programs equivalent if they describe the same trace language. This notion of equivalence is very strong as it requires equivalent programs to be able to invoke the same action sequences. Regardless of a possible interpretation of actions, equivalent programs behave the same. Consequently showing program transformations to be equivalence preserving ensures that the transformation preserves the observable behaviour.

To be able to use program equivalence for verification, we need to prove it a congruence relation with respect to the syntactic structure of programs.

Two programs s and t are called **equivalent** if they have the same traces. We write $\boldsymbol{s \approx t}$ if $\mathcal{T}s = \mathcal{T}t$ and $\boldsymbol{s \preccurlyeq t}$ if $\mathcal{T}s \subseteq \mathcal{T}t$.

Fact 3.1 $s \approx t$ is an equivalence relation such that $s \approx t$ iff $s \preccurlyeq t$ and $t \preccurlyeq s$.

Fact 3.2 (Important equivalences)

$$
\begin{aligned}
(s+t)+u &\approx s+(t+u) && \text{(Associativity)}\\
s+t &\approx t+s && \text{(Commutativity)}\\
\varnothing + s &\approx s && \text{(Left Identity)}\\
s+s &\approx s && \text{(Idempotence)}\\
s(tu) &\approx (st)u && \text{(Associativity)}\\
1s &\approx s && \text{(Left Identity)}\\
s1 &\approx s && \text{(Right Identity)}\\
s(t+u) &\approx st+su && \text{(Left Distributivity)}\\
(s+t)u &\approx su+tu && \text{(Right Distributivity)}\\
\varnothing s &\approx \varnothing && \text{(Left Annulation)}
\end{aligned}
$$

The equivalences of Fact 3.2 are well known from Kleene algebras [6]. The first four equations say that $(\mathsf{cfp}, +, \varnothing)$ is an idempotent commutative monoid. The next three equations say that $(\mathsf{cfp}, ;, 1)$[1] is a monoid. In summary, $(\mathsf{cfp}, +, ;)$ is an idem-

[1] We use ; here to denote the operator for sequential composition.

potent semi-ring without right annulating 0.

The only missing property for an idempotent semi-ring that is the right annulation of $\varnothing$.

Lemma 3.3 $x\varnothing \not\approx \varnothing$

Proof By definition, $x\epsilon$ is a trace of $x\varnothing$, but by Fact 2.6 $x\epsilon$ is no trace of $\varnothing$. ∎

From a computational point of view, the right composition of $\varnothing$ to a program s means that the program diverges silently after the execution of s. As s may have invoked actions before diverging silently, this actions are still recorded in partial traces.

3.1 Substitutivity

We assume a **parallel substitution operator** $\boldsymbol{s[\sigma]}$ for programs. Given a substitution $\sigma : \mathbb{N} \to \mathsf{cfp}$ (where cfp denotes the set of all context-free programs), $s[\sigma]$ yields the program obtained from s by synchronously replacing all occurrences of free variables x in s by σx. We call the set of all variables x with $\sigma x \neq x$ the **domain of a substitution** σ and write $\boldsymbol{\mathcal{D}\sigma}$. We write $\boldsymbol{s^x_t}$ for $s[\sigma]$ with $\mathcal{D}\sigma = \{x\}$ and $\sigma x = t$.

As usual we assume capture-avoiding substitution.

We call two substitutions σ and τ equivalent ($\boldsymbol{\sigma \approx \tau}$) if they agree on all variables up to equivalence ($\forall x.\, \sigma x \approx \tau x$). Analogously, we write $\boldsymbol{\sigma \preccurlyeq \tau}$ if $\mathcal{T}(\sigma x) \subseteq \mathcal{T}(\tau x)$ for all variables x.

We will show that $\approx$ is substitutive:

$$\frac{s \approx t \qquad \sigma \approx \tau}{s[\sigma] \approx t[\tau]}$$

Lemma 3.4 If $\sigma \preccurlyeq \tau$, then $s[\sigma] \preccurlyeq s[\tau]$

Proof Assume $\sigma \preccurlyeq \tau$ and $\xi/s[\sigma]$. The proof follows by induction on $\xi/s[\sigma]$.

Remark. Technically, the induction on $\xi/s[\sigma]$ does not go through as smoothly as expected. As the induction is on the structure of $s[\sigma]$ rather than on the structure of s, we get inductive hypotheses for programs t of the form $t = s[\sigma]$. To reconstruct the structure of s, we need inversion lemmas for each language construct. We have to distinguish the case where s has the same structure as t with the substitution applied to its components and the case where s was a variable x such that $t = \sigma x$. As the case that s is a variable, has to be considered in each case of the induction, proofs with inversion lemmas might grow unnecessarily large. An alternative is the introduction of a new trace predicate for programs under substitution that respects the structure of s. We used such an additional predicate to make our proofs shorter.

∎

Lemma 3.5 If $\sigma \approx \tau$, then $s[\sigma] \approx s[\tau]$.

Proof Follows with Lemma 3.4. ∎

We now show that substitution preserves equivalence. This requires a careful study of the traces of $s[\sigma]$. We introduce a predicate $\eta/\xi, \sigma$ that provides the following characterization of the traces of $s[\sigma]$.

$$\eta/s[\sigma] \leftrightarrow \exists \xi \in \mathcal{T}s.\ \eta/\xi, \sigma$$

The **substitution predicate $\eta/\xi, \sigma$** defines the traces η that can be obtained from a trace ξ by substituting traces of σx for variables $x \in \mathcal{V}\xi$.

$$\frac{}{\epsilon/\epsilon, \sigma} \qquad \frac{}{\#/\#, \sigma} \qquad \frac{\zeta/(\sigma\, x) \quad \eta/\xi, \sigma}{\zeta\, \eta/x\xi, \sigma}$$

Fact 3.6 $\exists \eta.\eta/\xi, \sigma$

We refer to $\eta/\xi, \sigma$ as **trace substitution**. Trace substitution is consistent with trace concatenation.

Fact 3.7

1. If $\eta_1/\xi_1, \sigma$ and $\eta_2/\xi_2, \sigma$, then $(\eta_1\, \eta_2)/(\xi_1\, \xi_2), \sigma$.
2. If $\eta/(\xi_1\, \xi_2), \sigma$, then there are traces η_1 and η_2 such that $\eta = \eta_1\, \eta_2$ and $\eta_1/\xi_1, \sigma$ and $\eta_2/\xi_2, \sigma$.

Every trace of a program s under a substitution σ can be seen as a trace resulting from a trace substitution of a trace ξ of s with σ.

Lemma 3.8 If ξ/s and $\eta/\xi, \sigma$, then $\eta/s[\sigma]$

Proof Induction on ξ/s. The case for sequential composition uses Fact 3.7. ∎

Lemma 3.9 (Trace Decomposition) If $\eta/s[\sigma]$, then there is a trace ξ such that ξ/s and $\eta/\xi, \sigma$.

Proof Induction on $\xi/s[\sigma]$. The case for sequential composition uses Fact 3.7. ∎

Substitution preserves program subsumption.

Lemma 3.10 If $s \preccurlyeq t$, then $s[\sigma] \preccurlyeq t[\sigma]$.

Proof Assume $s \preccurlyeq t$. Let η be a trace of $s[\sigma]$. By Lemma 3.9, η can be decomposed into a trace ξ of s such that $\eta/\xi, \sigma$. By the assumption, we get that ξ is a trace of t. We obtain $\eta/t[\sigma]$ using Lemma 3.8 and $\eta/\xi, \sigma$. ∎

As a consequence, substitution preserves program equivalence.

Lemma 3.11 If $s \approx t$, then $s[\sigma] \approx t[\sigma]$.

Proof By Lemma 3.10. ∎

Theorem 3.12 (Substitutivity) If $s \approx t$ and $\sigma_1 \approx \sigma_2$, then $s[\sigma_1] \approx t[\sigma_2]$.

Proof Follows with Lemmas 3.5 and 3.11. ∎

3.2 Congruence

We now prove that $s \approx t$ is a congruence relation. The proof of the congruence law for recursion

$$\frac{s \approx t}{\mu x.s \approx \mu x.t}$$

relies on a detailed analysis of unfolding.

We define a **unfolding operator** $[s]^{\boldsymbol{n}}_{\boldsymbol{x}}$ limiting the number of substitutions made in the recursion case by a **bound** n.

The unfolding operator allows us to characterize the traces of $\mu x.s$ as follows:

$$\mathcal{T}(\mu x.s) = \bigcup_{n \in \mathbb{N}} \mathcal{T}[s]^n_x$$

Definition 3.13

$$\begin{aligned} &[\cdot]^n_x : \mathsf{cfp} \to \mathbb{N} \to \mathsf{var} \to \mathsf{cfp} \\ &[s]^0_x = \varnothing \\ &[s]^n_x = s^x_{[s]^{n-1}_x} \qquad\qquad n > 0 \end{aligned}$$

Fact 3.14 (Monotonicity) If $n \le m$ then $[s]^n_x \preccurlyeq [s]^m_x$.

Unfolding and substitution commute.

Fact 3.15 If $x \notin \mathcal{D}\sigma$, then $([s]^n_x)[\sigma] = [s[\sigma]]^n_x$.

The semantics of recursion can be characterized using the unfolding operator.

Lemma 3.16 If $\xi / s^x_{\mu x.t}$, then $\xi / s^x_{[t]^n_x}$ for some n.

Proof Induction on $\xi / s^x_{\mu x.t}$. All cases follow directly from the inductive hypotheses, except those for sequential composition and recursion.

- $s = s_1 s_2$. By the inductive hypotheses we know that there are ξ_1 and ξ_2 such that $\xi = \xi_1\xi_2$ and $\xi_1/s_1{}^{x}_{[t]^{n_1}_x}$ and $\xi_2/s_2{}^{x}_{[t]^{n_2}_x}$ for some n_1, n_2. We show that $\xi_1\xi_2/(s_1 s_2)^{x}_{[t]^{n_1+n_2}_x}$. By definition, it is sufficient to show that $\xi_1/[s_1]^{n_1+n_2}_x$ and $\xi_2/[s_2]^{n_1+n_2}_x$. This follows directly using Lemma 3.4, the monotonicity of the unfolding operator (Fact 3.14), and the inductive hypotheses.
- $s = \mu y.s$. We distinguish two cases:
  1. $x = y$. Then $(\mu y.s)^{x}_{\mu x.t} = \mu y.s = (\mu y.s)^{x}_{[t]^{n}_x}$. Consequently the statement holds trivially.
  2. $x \neq y$. By the inductive hypothesis we know that $\xi/(s^{y}_{\mu y.s})^{x}_{[t]^{n}_x}$ for some n. As $x \neq y$ and $[t]^n_x$ does not contain free occurrences of x, we can conclude that $\xi/(s^{x}_{[t]^{n}_x})^{y}_{\mu y.s^{x}_{[t]^{n}_x}}$. Hence $\xi/\mu y.s^{x}_{[t]^{n}_x}$ and also $\xi/(\mu y.s)^{x}_{[t]^{n}_x}$. ∎

Lemma 3.17 (Correctness of unfolding) $\xi/\mu x.s$ iff $\xi/[s]^n_x$ for some n.

Proof We show the two directions of the proof separately:

- Assume $\xi/\mu x.s$. Then there are two possibilities for ξ:
  1. $\xi = \epsilon$. Then ξ is a trace of $\varnothing = [s]^0_x$.
  2. $\xi/s^{x}_{\mu x.s}$. By Lemma 3.16, we know that there is an $n \in \mathbb{N}$ such that $\xi/s^{x}_{[s]^{n}_x}$. Consequently, $\xi/[s]^{n+1}_x$ by Definition 3.13.
- Assume $\xi/[s]^n_x$ for some $n \in \mathbb{N}$. Proof by induction on $n \in \mathbb{N}$.
  - $n = 0$. Then $\xi/\varnothing$ and consequently $\xi = \epsilon$ (by Fact 2.6). Hence $\xi/\mu x.s$ (Fact 2.7).
  - $n > 0$. Then $[s]^n_x = s^{x}_{[s]^{n-1}_x}$. With the inductive hypothesis for $n-1$ and Lemma 3.4, we get that $\xi/s^{x}_{\mu x.s}$. By definition, we have $\xi/\mu x.s$. ∎

The following lemma gives us the compatibility of $\approx$ with recursion as a corollary.

Lemma 3.18 If $s \preccurlyeq t$ then $[s]^n_x \preccurlyeq \mu x.t$

Proof Assume $s \preccurlyeq t$ and $\xi/[s]^n_x$. Proof by induction on $n \in \mathbb{N}$.

- $n = 0$. Then $\xi/\varnothing$ and consequently (Fact 2.6) $\xi = \epsilon$. Hence $\xi/\mu x.t$ (Fact 2.7).
- $n > 0$. Then $\xi/s^{x}_{[s]^{n-1}_x}$. With the inductive hypothesis and Lemma 3.4, we can conclude that $\xi/s^{x}_{\mu x.t}$. Then we also have $\xi/t^{x}_{\mu x.t}$ (Lemma 3.10) and consequently $\xi/\mu x.t$. ∎

Theorem 3.19 (Congruence) Program equivalence is an equivalence relation compatible with the syntactic structure of programs:

1. $s \approx s' \to t \approx t' \to st \approx s't'$
2. $s \approx s' \to t \approx t' \to s + t \approx s' + t'$
3. $s \approx s' \to \mu x.s \approx \mu x.s'$

Proof The first two rules follow directly from the definitions. The congruence rule for recursion follows with Lemma 3.18 and the correctness of unfolding (Lemma 3.17). ∎

Corollary 3.20 Program equivalence is a substitutive congruence.

Proof Follows with Theorem 3.12 and Theorem 3.19 ∎

Chapter 4

Tail-Recursion and Regularity

We will focus on programs using tail-recursion only. Tail-recursive programs can easily be translated to jumps in register transfer languages [7].

We compare tail-recursive programs with regular programs. Regular programs are programs where recursion is restricted to iteration and the null program. We will show that tail-recursive programs and regular programs can describe the same trace languages. To do so, we construct a translator from tail-recursive to regular programs and show its correctness.

4.1 Tail-Recursion

A program is *tail-recursive* if all occurrences of bound variables are in *tail position*. A variable is in tail position in a program, if it only occurs in the final component of a sequential composition. Examples for tail-recursive programs are:

$$1 \qquad x \qquad \mu x.x \qquad (\mu x.x)y \qquad \mu x.1 + ax \qquad \mu x.x + (\mu y.x + y)$$

Tail-recursion is inductively defined with respect to a set V of bound variables. We use the notation $A \parallel B$ to say that two sets A and B are disjoint.

$$\frac{}{\mathsf{trec}_V\ 1} \qquad \frac{}{\mathsf{trec}_V\ x} \qquad \frac{V \parallel \mathcal{V}s \qquad \mathsf{trec}_\emptyset\ s \qquad \mathsf{trec}_V\ t}{\mathsf{trec}_V\ (st)} \qquad \frac{\mathsf{trec}_V\ s \qquad \mathsf{trec}_V\ t}{\mathsf{trec}_V\ (s + t)}$$

$$\frac{\mathsf{trec}_{(V \cup \{x\})}\ s}{\mathsf{trec}_V\ (\mu x.s)}$$

We say that s is ***V*-tail-recursive** if $\mathsf{trec}_V\ s$. If s is V-tail-recursive all bound variables as well as the variables in V are located in tail position.

Fact 4.1 (Weakening) If $\mathsf{trec}_V\ s$ and $U \subseteq V$, then $\mathsf{trec}_U\ s$.

Fact 4.2 (Strengthening) If $\mathsf{trec}_V\ s$ and $U \parallel \mathcal{V}s$, then $\mathsf{trec}_{V \cup U}\ s$.

Fact 4.3 (Substitution) Let $\mathsf{trec}_V\ s$, $x \in V$, and $\mathsf{trec}_V\ t$. Then $\mathsf{trec}_V\ s^x_t$.

4.2 Regularity

Regular programs restrict recursion to iterations and the null program (Definition 2.5). Regular programs can therefore be seen as the generalization of programs with while loops such as IMP [13]. In the following we will develop a formal notion for the regularity of programs.

We define **iteration** (Kleene iteration) for programs.

$$s^* := \mu x.1 + sx \qquad\qquad x \text{ not free in } s$$

A program is called *regular* if all recursions are of the form $\varnothing$ or s^*.

Accordingly, we can define regularity inductively.

$$\frac{}{\mathsf{reg}\,1} \qquad \frac{}{\mathsf{reg}\,x} \qquad \frac{\mathsf{reg}\,s \qquad \mathsf{reg}\,t}{\mathsf{reg}\,(st)} \qquad \frac{\mathsf{reg}\,s \qquad \mathsf{reg}\,t}{\mathsf{reg}\,(s+t)} \qquad \frac{x \notin \mathcal{V}s \qquad \mathsf{reg}\,s}{\mathsf{reg}\,(\mu x.1+sx)} \qquad \frac{}{\mathsf{reg}\,\varnothing}$$

We give another definition of regularity corresponding to the structure of the predicate $\mathsf{trec}_V\, s$. We define regularity with respect to a set V of variables.

$$\frac{}{\mathsf{reg}_V\,1} \qquad \frac{x \notin V}{\mathsf{reg}_V\,x} \qquad \frac{\mathsf{reg}_V\,s \qquad \mathsf{reg}_V\,t}{\mathsf{reg}_V\,(st)} \qquad \frac{\mathsf{reg}_V\,s \qquad \mathsf{reg}_V\,t}{\mathsf{reg}_V\,(s+t)} \qquad \frac{}{\mathsf{reg}_V\,\varnothing}$$

$$\frac{\mathsf{reg}_{(V\cup\{x\})}\,s}{\mathsf{reg}_V\,(\mu x.1+sx)}$$

We call a program s **V-regular** if $\mathsf{reg}_V\, s$. If s is V-regular, s is regular and does not contain any free occurrences of variables in V.

Fact 4.4 If $\mathsf{reg}_V\, s$, then $\mathcal{V}s \parallel V$.

Fact 4.5 (Weakening) If $\mathsf{reg}_V\, s$ and $U \subseteq V$, then $\mathsf{reg}_U\, s$.

Fact 4.6 (Strengthening) If $\mathsf{reg}_V\, s$ and $U \parallel \mathcal{V}s$, then $\mathsf{reg}_{(V\cup U)}\, s$.

The predicate $\mathsf{reg}_V\, s$ can characterized in terms of $\mathsf{reg}\, s$.

Lemma 4.7 Let $\mathsf{reg}_V\, s$. Then $\mathsf{reg}\, s$.

Proof Induction on $\mathsf{reg}_V\, s$. Only the case for iteration (s^*) requires to show that $x \notin \mathcal{V}s$. This follows directly by $\mathsf{reg}_{V\cup\{x\}}\, s$ and Fact 4.4. ∎

Lemma 4.8 Let $\mathsf{reg}\, s$ then $\mathsf{reg}_\emptyset\, s$.

Proof Induction on $\mathsf{reg}\, s$. Only the case for iteration (s^*) requires to show that $\mathsf{reg}_{\{x\}}\, s$. This follows directly from $x \notin \mathcal{V}s$, Fact 4.6 and the inductive hypothesis. ∎

Lemma 4.9 $\mathsf{reg}_V\, s$ iff $\mathsf{reg}\, s$ and $\mathcal{V}s \parallel V$.

Proof The direction from left to right follow directly by Fact 4.4 and Lemma 4.7. The direction from right to left follows directly from Lemma 4.8 and Fact 4.6. ∎

4.3 Correspondence of Tail-Recursion and Regularity

Tail-recursive programs can describe the same class of trace languages as regular programs. Regular programs only allow a restricted form of tail-recursion. Consequently, every regular program is tail-recursive. Moreover, every tail-recursive program can be translated into an equivalent regular program.

In contrast to tail-recursive programs, in regular programs all recursions are *short distance*. Short distance recursions are recursions where a variable bound by μ only occurs in the top-level program of the recursion. Examples for short distance recursions are:

$$\mu x.x \qquad \mu x.1 \qquad \mu x.1 + ax \qquad \mu x.(x + \mu y.y)$$

In contrast, the outer recursion of the following program is not short distance:

$$\mu x.(x + \mu y.x + y)$$

We call recursion that are not short distance *long distance*. The translation of tail-recursive programs to regular programs can therefore be seen as the elimination of long distance recursion.

Theorem 4.10 If $\mathsf{reg}_V\ s$, then $\mathsf{trec}_V\ s$.

Proof Induction on $\mathsf{reg}_V\ s$ using Fact 4.4. ∎

To show that every tail-recursive program can be translated into a regular program, we need to show how tail recursions can be translated into iterations. It is well-known that this transformation is possible [9].

We show that recursions of the form $\mu x.s + tx$ can easily be transformed into iterations if x is not free in s and t. We call recursions of this form **decomposed**.

Decomposed recursions satisfy the following unfolding property:

Fact 4.11 Let $x \notin \mathcal{V}s \cup \mathcal{V}t$. Then $[s + tx]_x^{n+1} \approx s + t[s + tx]_x^n$.

Decomposed recursions satisfy a distributivity law.

Lemma 4.12 Let $x \notin \mathcal{V}s \cup \mathcal{V}t \cup \mathcal{V}u$. Then $[s + tx]_x^n u \approx [su + tx]_x^n$

Proof Proof by induction on $n \in \mathbb{N}$.

- $n = 0$. Then $[s + tx]_x^0 u = \varnothing u \approx \varnothing = [su + tx]_x^0$. (Fact 3.2)
- $n > 0$. Then

$$\begin{aligned}
[s + tx]_x^n u &\approx (s + t[s + tx]_x^{n-1})u && \text{Fact 4.11} \\
&\approx su + [s + tx]_x^{n-1} u && \text{Distributivity} \\
&\approx su + [su + tx]_x^{n-1} && \text{Inductive hypothesis} \\
&\approx [su + tx]_x^n && \text{Fact 4.11}
\end{aligned}$$

∎

Corollary 4.13 (Distributivity) Let $x \notin \mathcal{V}s \cup \mathcal{V}t \cup \mathcal{V}u$. Then

$$(\mu x.s + tx)u \approx \mu x.su + tx$$

Proof Follows using Lemmas 3.17 and 4.12. ∎

Corollary 4.14 If $x \notin \mathcal{V}s \cup \mathcal{V}t$ then $\mu x.s + tx \approx t^*s$.

Proof Assume $x \notin \mathcal{V}s \cup \mathcal{V}t$. Then $t^*s = (\mu x.1 + tx)s \approx \mu x.1s + tx \approx \mu x.s + tx$ using Corollary 4.13. ∎

4.3.1 Regularizer

The regularization of a program consists of two parts:

- Recursions with regular bodies are translated to decomposed recursions $\mu x.s + tx$ with regular components s and t. We call such decomposed recursions *regular*.
- Regular decomposed recursions are translated to regular programs.

The general idea of regularization is that recursions are resolved in a bottom-up manner. The inner most recursions have regular bodies and can therefore be transformed to regular decomposed recursions. Such regular decomposed recursions can by Corollary 4.14 be translated to regular programs. By a step-wise eliminating recursions, the whole program can be regularized.

The **decomposer** $\boldsymbol{D}$ is a function that for a variable x decomposes a regular program s into two regular programs s_1 and s_2 such that $s \approx s_1 + s_2x$ and s_1 and s_2 satisfying the constraints of Corollary 4.14. Regularity and V-tail-recursion are preserved by the decomposition. The decomposer allows us to transform recursions with regular bodies to regular decomposed recursions.

$$\begin{aligned}
D &: \mathsf{var} \to \mathsf{cfp} \to \mathsf{cfp} \times \mathsf{cfp} \\
D_x\, 1 &:= (1, \varnothing) \\
D_x\, x &:= (\varnothing, 1) \\
D_x\, y &:= (y, \varnothing) && x \neq y \\
D_x\, (st) &:= \mathsf{let}\, (t_1, t_2) := D_x\, t\, \mathsf{in}\, (st_1, st_2) \\
D_x\, (s + t) &:= \mathsf{let}\, (s_1, s_2) := D_x\, s; (t_1, t_2) := D_x\, t\, \mathsf{in}\, (s_1 + t_1, s_2 + t_2) \\
D_x\, (\mu y.s) &:= (\mu y.s, \varnothing)
\end{aligned}$$

Lemma 4.15 If $\mathsf{trec}_V\ s$ and $(s_1, s_2) = D_x\, s$, then $s \approx s_1 + s_2x$.

Proof By induction on $\mathsf{trec}_V\ s$. ∎

Lemma 4.16 Let $\mathsf{trec}_{(V\cup\{x\})}\, s$ and $\mathsf{reg}\, s$. Let $(s_1, s_2) = D_x\, s$. Then

1. $\mathsf{reg}_{\{x\}}\, s_1$
2. $\mathsf{trec}_V\, s_1$
3. $\mathsf{reg}_{(V\cup\{x\})}\, s_2$
4. $\mathcal{V}s - \{x\} = \mathcal{V}s_1 \cup \mathcal{V}s_2$

Proof By induction on $\mathsf{trec}_{(V\cup\{x\})}\, s$. ∎

Theorem 4.17 (Correctness) Let s be regular and $(V \cup \{x\})$-tail-recursive. Let $(s_1, s_2) = D_x\, s$. Then

1. $\mu x.s \approx s_2^* s_1$.
2. $s_2^* s_1$ is V-tail-recursive and regular.
3. $\mathcal{V}(\mu x.s) = \mathcal{V}(s_2^* s_1)$.

Proof

1. We have $s \approx s_1 + s_2 x$ by Lemma 4.15. By Lemma 4.16, we know $\mathsf{reg}_{\{x\}}\, s_1$ and $\mathsf{reg}_{V\cup\{x\}}\, s_2$. With Fact 4.4 we conclude that $x \notin \mathcal{V}s_1$ and $x \notin \mathcal{V}s_2$. Using Corollary 4.14, we get $\mu x.s \approx \mu x.s_1 + s_2 x \approx s_2^* s_1$.
2. Lemma 4.16 gives us $\mathsf{reg}_{\{x\}}\, s_1$ and $\mathsf{reg}_{V\cup\{x\}}\, s_2$. From $\mathsf{reg}_{V\cup\{x\}}\, s_2$ we conclude $\mathsf{reg}_V\, s_2^*$. By Lemma 4.9, we get $\mathsf{reg}\, s_1$ and $\mathsf{reg}\, s_2^*$ and consequently $\mathsf{reg}\, s_2^* s_1$. In addition we conclude the following from $\mathsf{reg}_V\, s_2^*$:
   (a) $\mathsf{trec}_V\, s_2^*$ (Theorem 4.10) and using weakening (Fact 4.1) also $\mathsf{trec}_\emptyset\, s_2^*$.
   (b) $\mathcal{V}(s_2^*) \parallel V$ (Fact 4.4).
   As Lemma 4.16 gives us $\mathsf{trec}_V\, s_1$, we conclude $\mathsf{trec}_V\, s_2^* s_1$.
3. From Lemma 4.16 we get $\mathsf{reg}_{\{x\}}\, s_1$ and consequently know that $x \notin \mathcal{V}s_1$ (Fact 4.4). In addition, Lemma 4.16 gives us $\mathcal{V}s - \{x\} = \mathcal{V}s_1 \cup \mathcal{V}s_2$. Then:

$$\begin{aligned}
\mathcal{V}(s_2^* s_1) &= \mathcal{V}s_2^* \cup \mathcal{V}s_1 && \text{Definition of } \mathcal{V}\\
&= (\mathcal{V}s_2 - \{x\}) \cup \mathcal{V}s_1 && \text{Definition of } \mathcal{V}\\
&= (\mathcal{V}s_2 \cup \mathcal{V}s_1) - \{x\} && \text{As } x \notin \mathcal{V}s_1\\
&= \mathcal{V}s - \{x\} && \text{As } \mathcal{V}s - \{x\} = \mathcal{V}s_1 \cup \mathcal{V}s_2\\
&= \mathcal{V}(\mu x.s) && \text{Definition of } \mathcal{V}
\end{aligned}$$

∎

We define a function R constructing an equivalent regular program from a tail-recursive program using the decomposer. R is called **regularizer**.

$$\begin{aligned} R &: \mathsf{cfp} \to \mathsf{cfp} \\ R\,1 &:= 1 \\ R\,x &:= x \\ R\,(st) &:= (R\,s)(R\,t) \\ R\,(s+t) &:= R\,s + R\,t \\ R\,(\mu x.s) &:= \mathsf{let}\,(s_1, s_2) := D_x\,(R\,s)\,\mathsf{in}\,s_2^* s_1 \end{aligned}$$

The regularizer satisfies three properties:

1. preservation of tail-recursion
2. preservation of free variables
3. establishment of regularity

Lemma 4.18 Let $\mathsf{trec}_V\ s$. Then the following holds:

1. $\mathsf{trec}_V\,(R\,s)$
2. $\mathcal{V}s = \mathcal{V}(R\,s)$
3. $\mathsf{reg}\,R\,s$

Proof Induction on $\mathsf{trec}_V\ s$. All cases are easy except the cases for sequential composition and recursion.

- $s = s_1 s_2$. Then $\mathcal{V}s_1 \parallel V$ and also $\mathcal{V}(R\,s_1) \parallel V$ by the inductive hypothesis. By the inductive hypotheses for for s_1 and s_2, we have $\mathsf{trec}_\emptyset\,(R\,s_1)$ and $\mathsf{trec}_V\,(R\,s2)$. We conclude $\mathsf{trec}_V\,((R\,s_1)(R\,s_2))$. The regularity of $(R\,s_1)(R\,s_2)$ and $\mathcal{V}(s_1 s_2) = \mathcal{V}((R\,s_1)(R\,s_2))$ follows directly from the inductive hypotheses.
- $s = \mu x.s$. As $R\,s$ is $(V \cup \{x\})$-tail-recursive and regular (by inductive hypothesis), we know for the decomposition $(s_1, s_2) = D_x\,(R\,s)$ that $s_2^* s_1$ is V-tail-recursive and regular (Theorem 4.17). In addition:

$$\begin{aligned} \mathcal{V}(\mu x.s) &= \mathcal{V}s - \{x\} && \text{Definition of } \mathcal{V} \\ &= \mathcal{V}(R\,s) - \{x\} && \text{Inductive hypothesis for } s \\ &= \mathcal{V}(\mu x.R\,s) && \text{Definition of } \mathcal{V} \\ &= \mathcal{V}(s_2^* s_1) && \text{Theorem 4.17} \end{aligned}$$

■

Lemma 4.19 If $\mathsf{trec}_V\ s$, then $R\,s \approx s$.

Proof Induction on $\mathsf{trec}_V\ s$. All cases are easy but $\mu x.s$. By the inductive hypothesis we know that $s \approx R\,s$ and consequently $\mu x.s \approx \mu x.(R\,s)$. By Lemma 4.18, we obtain $\mathsf{trec}_{V\cup\{x\}}\,(R\,s)$ and $\mathsf{reg}\,(R\,s)$. The correctness of the decomposer (Theorem 4.17) yields $\mu x.s \approx \mu x.(R\,s) \approx s_2^* s_1$ for the decomposition $(s_1, s_2) = D_x\,(R\,s)$. ■

Theorem 4.20 (Correctness) Let s be V-tail-recursive. Then $R\,s$ is regular, V-tail-recursive, and equivalent to s.

Proof Using Lemmas 4.18 and 4.19. ∎

Chapter 5

Linear Tail-Recursive Programs

Tail-recursive programs as well as regular programs allow full sequential compositions. Full sequential compositions can cause the situation that different execution paths of a program need to continue with the same continuation. To remember the continuation a stack is needed. The need for a stack can be avoided by restricting sequential composition such that only actions occur as first component. Programs satisfying this constraint are called *linear*.

We show that every tail-recursive program can be translated into an equivalent linear tail-recursive program. To do so, we give a translation from regular programs to linear tail-recursive programs that can be verified locally using the equivalences from Fact 3.2.

In addition we study a direct compiler from tail-recursive programs to linear tail-recursive programs. A direct translation from tail-recursive programs to linear programs, as it is needed in practice, cannot be verified locally. The correctness proof requires a careful study of the traces of tail-recursive programs. We give such a direct compiler and verify its correctness.

5.1 Linearity

A program is *linear* if all sequential compositions are of the form xs. We call programs which are linear and tail-recursive *linear tail-recursive*. We define linear tail-recursion by an inductive predicate with respect to a set V of variables.

$$\frac{}{\mathsf{ltrec}_V\ 1} \qquad \frac{}{\mathsf{ltrec}_V\ x} \qquad \frac{x \notin V \qquad \mathsf{ltrec}_V\ s}{\mathsf{ltrec}_V\ (xs)} \qquad \frac{\mathsf{ltrec}_V\ s \qquad \mathsf{ltrec}_V\ t}{\mathsf{ltrec}_V\ (s+t)} \qquad \frac{\mathsf{ltrec}_{(V\cup\{x\})}\ s}{\mathsf{ltrec}_V\ (\mu x.s)}$$

We say that s is **V-linear tail-recursive** if $\mathsf{ltrec}_V\ s$.

Fact 5.1 (Weakening) If $\mathsf{ltrec}_V\ s$ and $U \subseteq V$, then $\mathsf{ltrec}_U\ s$.

Fact 5.2 (Strengthening) If $\mathsf{ltrec}_V\ s$ and $U \parallel \mathcal{V}s$ then $\mathsf{ltrec}_{V\cup U}\ s$.

Fact 5.3 If $\mathsf{ltrec}_V\ s$, then $\mathsf{trec}_V\ s$.

5.2 Linearizing regular programs

In the last chapter we defined and verified a compiler from tail-recursive programs to regular programs. Consequently, a compiler from regular programs to linear programs suffices to show the linearizability of tail-recursive programs.

Regular programs have the advantage that all recursions are short distance. For this reason the compiler does not need to track the bound variables of the program. This structure simplifies the translation from regular programs to linear tail-recursive programs.

The main challenge of linearization is to dissolve sequential compositions. The following equivalences describe how sequential compositions can be dissolved for the different language constructs as left component.

$$\begin{aligned} 1u &\approx u \\ (st)u &\approx s(tu) \\ (s+t)u &\approx su + tu \\ (\varnothing)u &\approx \varnothing \\ (\mu x.1 + sx)u &\approx \mu x.u + sx \qquad x \notin \mathcal{V}s \cup \mathcal{V}u \end{aligned}$$

All equivalences displayed here are known from Fact 3.2 and Corollary 4.13. The compiler for regular expressions (that we call *linearizer*) applies these equivalences recursively to generate a linearized program. Consequently, the linearizer can be verified locally using these equivalences.

The linearizer for regular programs is defined with respect to a *continuation* u.

$$\begin{aligned} L &: \mathsf{cfp} \to \mathsf{cfp} \to \mathsf{cfp} \\ L\,1\,u &:= u \\ L\,x\,u &:= xu \\ L\,(st)\,u &:= L\,s\,(L\,t\,u) \\ L\,(s+t)\,u &:= L\,s\,u + L\,t\,u \\ L\,\varnothing\,u &:= \varnothing \\ L\,s^*\,u &:= \mu x.u + L\,s\,x \qquad x \notin \mathcal{V}u \end{aligned}$$

Lemma 5.4 Let $\mathsf{reg}_V\ s$ and $\mathsf{lin}\,Vu$. Then $\mathsf{lin}\,V(L\,s\,u)$.

Proof By induction on $\mathsf{reg}_V\ s$. ∎

Lemma 5.5 Let $\mathsf{reg}\,s$. Then $L\,s\,u \approx su$.

Proof Induction on $\mathsf{reg}\,s$. All cases follow directly with the inductive hypotheses and the equivalences from Fact 3.2. The recursion case uses Corollary 4.14. ∎

Corollary 5.6 (Correctness) Let s be V-regular and u be V-linear tail-recursive. Then $L\,s\,u$ is V-linear tail-recursive and equivalent to su.

Proof Using Lemmas 5.4 and 5.5. ∎

5.3 Linearizing tail-recursive programs

It is well-known that tail-recursive programs can be translated directly to programs of register transfer languages using jumps [7]. Translating tail-recursive programs to regular programs first, consequently causes an unnecessary restructuring of the program. The linearizer discussed in this section is more straightforward as it performs a direct translation respecting the structure of the program.

We define a function $\boldsymbol{L}$ that translates a tail-recursive program to an equivalent tail-recursive linear program. We call L *linearizer*. The linearizer L is formulated with respect to a continuation u.

$$
\begin{aligned}
L &: \mathsf{var}\,\mathit{set} \to \mathsf{cfp} \to \mathsf{cfp} \to \mathsf{cfp} \\
L_V\,1\,u &:= u \\
L_V\,x\,u &:= \mathit{if}\ x \in V\ \mathit{then}\ x\ \mathit{else}\ xu \\
L_V\,(st)\,u &:= L_V\,s\,(L_V\,t\,u) \\
L_V\,(s+t)\,u &:= L_V\,s\,u + L_V\,t\,u \\
L_V\,(\mu x.s)\,u &:= \mu x.L_{V\cup\{x\}}\,s\,u \qquad\qquad x \notin \mathcal{V}u
\end{aligned}
$$

Note that the linearizer shown here would produce an exponential overhead in the choice case. This could be easily avoided in practice by introducing a let-construct. As we want the language to remain as simple as possible we do not use such constructs here.

The linearizer preserves tail-recursion.

Lemma 5.7 (Preservation of tail-recursion)
Let $\mathsf{trec}_V\,s$ and $\mathsf{trec}_V\,u$. Then $\mathsf{trec}_V\,(L_V\,s\,u)$.

Proof By induction on $\mathsf{trec}_V\,s$. ∎

The linearizer establishes linear tail-recursion.

Lemma 5.8 (Establishment of linearity)
Let $\mathsf{trec}_V\,s$ and $\mathsf{ltrec}_V\,u$. Then $\mathsf{ltrec}_V\,(L_V\,s\,u)$.

Proof By induction on $\mathsf{trec}_V\,s$. The recursion case uses Fact 4.3. ∎

We will show that the linearizer satisfies the following equation:

$$L_\emptyset\,s\,u \approx su$$

5.3.1 Traces of tail-recursive programs under substitution

The correctness proof requires a careful study of the traces of V-tail-recursive programs. A V-tail-recursive programs s contains free variables $x \in V$ in tail position, this carries over to the traces of s. Consequently, variables $x \in V$ can occur as last elements of traces of s.

We say that a variable x is **terminal** in ξ, if x is the last element of ξ and ξ does not contain x at another position. We also say that $\boldsymbol{\xi}$ **ends with** $\boldsymbol{x}$ if x is terminal in ξ. A trace ξ is $\boldsymbol{V}$**-tail-recursive** for a set V of variables, if either $\mathcal{V}\xi \parallel V$ or there is an variable $x \in V$ such that x is terminal in ξ and $\mathcal{V}\xi \cap V = \{x\}$.

Fact 5.9 Let ξ_1 be total, $x \notin \mathcal{V}\xi_1$, and x be terminal in ξ_2. Then x is terminal in $\xi_1\xi_2$.

Fact 5.10 Let x be terminal in $\xi_1\xi_2$, ξ_1 be total, and $x \notin \mathcal{V}\xi_1$. Then x is terminal in ξ_2.

Fact 5.11 Let x be terminal in $\xi_1\xi_2$. Then one of the following holds:

1. x is terminal in ξ_2 and ξ_1 is total.
2. x is terminal in ξ_1 and either ξ_1 is partial or ξ_2 has no elements.

Fact 5.12 Let $\mathsf{trec}_V\ s$ and ξ/s. Then ξ is V-tail-recursive.

With ξ^- we denote the trace ξ without its last element. For example we have:

$$\begin{aligned}\epsilon^- &= \epsilon\\ \#^- &= \#\\ (x\epsilon)^- &= \epsilon\\ (x\#)^- &= \#\\ (xy\epsilon)^- &= x\epsilon\end{aligned}$$

Fact 5.13 (Properties of element removal)

1. $\xi^-\epsilon = (\xi\epsilon)^-$
2. Let ξ_2 contain at least one element. Then $\xi_1\xi_2^- = (\xi_1\xi_2)^-$.

Fact 5.14 If x is terminal in ξ, then $x \notin \mathcal{V}\xi^-$.

Fact 5.15 If $\xi /\!\!/ s$, then $\xi^- /\!\!/ s$.

Fact 5.16 If ξ is V-tail-recursive, then $\mathcal{V}\xi^- \parallel V$.

We study the traces of tail-recursive programs under substitution.

To obtain an intuitive understanding of the following lemmas, we will argue about the execution paths of a program. Since the traces of a program are obtained from its possible execution paths, the changes of execution paths are reflected in the traces.

If none of the variables of the domain of a substitution σ appears on the execution path of a program s then the path stays unchanged under σ.

Lemma 5.17 If ξ/s and $\mathcal{D}\sigma \parallel \mathcal{V}\xi$, then $\xi/s[\sigma]$.

Proof Induction on ξ/s. All cases except sequential composition follow directly from the inductive hypotheses or by $\mathcal{D}\sigma \parallel \mathcal{V}\xi$.

Let $s = s_1 s_2$. Then there are traces ξ_1 and ξ_2 such that $\xi = \xi_1\xi_2$, ξ_1/s_1 and ξ_2/s_2. From $\mathcal{D}\sigma \parallel \mathcal{V}\xi_1\xi_2$ we can conclude that $\mathcal{D}\sigma \parallel \mathcal{V}\xi_1$ and therefore get from the inductive hypothesis for s_1 that $\xi_1/s_1[\sigma]$. As we only know that $\mathcal{D}\sigma \parallel \mathcal{V}\xi_2$, if ξ_1 is total (Fact 2.2), we distinguish whether ξ_1 is partial:

- ξ_1 partial : Then $\xi = \xi_1\xi_2 = \xi_1$ (Fact 2.1). As ξ_1 is partial, we also have $\xi_1/s_1[\sigma]s_2[\sigma]$ (Fact 2.4) and consequently $\xi/(s_1s_2)[\sigma]$.
- ξ_1 total: Then we conclude that $\mathcal{D}\sigma \parallel \mathcal{V}\xi_2$ and consequently get by the inductive hypotheses that $\xi_1/s_1[\sigma]$ and $\xi_2/s_2[\sigma]$ and hence also $\xi_1\xi_2/(s_1s_2)[\sigma]$. ∎

If an execution path of a program ends with a variable that will be substituted, it continues with the execution of the substituted program.

Lemma 5.18 Let ξ/s with $\mathcal{D}\sigma \cap \mathcal{V}\xi = \{x\}$ and x terminal in ξ. If $\eta/\sigma x$, then $\xi^-\eta/s[\sigma]$.

Proof Induction on ξ/s. All cases except the sequential composition follow directly from the inductive hypotheses. Let $s = s_1 s_2$. Then there are traces ξ_1, ξ_2 with ξ_1/s_1 and ξ_2/s_2 such that $\xi = \xi_1\xi_2$. Additionally we have that x is terminal in $\xi_1\xi_2$. This gives us two cases (Fact 5.11):

1. ξ_1 is a total trace not containing x and x is terminal in ξ_2. Then by inductive hypothesis for s_2 we get $\xi_2^-\eta/s[\sigma]$. As ξ_1 does not contain x we have $\mathcal{V}\xi_1 \parallel \mathcal{D}\sigma$. We conclude by Lemma 5.17 that $\xi_1/s[\sigma]$. Consequently $\xi_1\xi_2^-\eta/(s_1s_2)[\sigma]$. As ξ_2 contains at least one element (x), we know $\xi_1\xi_2^-\eta = (\xi_1\xi_2)^-\eta$ (Fact 5.13) and consequently $\xi^-\eta/(s_1s_2)[\sigma]$.
2. x is terminal in ξ_1 and ξ_1 is partial or ξ_2 contains no elements. We distinguish the different cases for ξ_1 and ξ_2.
   (a) ξ_1 is partial. Then $\xi = \xi_1\xi_2 = \xi_1$. As x is terminal in ξ_1 we conclude with the inductive hypothesis for s_1 that $\xi_1^-\eta/s_1[\sigma]$. As ξ_1^- is partial, $\xi_1^-\eta = \xi_1^-$ (Fact 2.1) and $\xi_1^-/s_1[\sigma]s_2[\sigma]$ (Fact 2.4). With $\xi = \xi_1$ and ξ_1^- being partial we finally get $\xi^-\eta/(s_1s_2)[\sigma]$.
   (b) $\xi_2 = \epsilon$. As $\epsilon/\sigma x$ we conclude with the inductive hypothesis for s_1 that $\xi_1^-\epsilon/s_1[\sigma]$. As $\xi_1^-\epsilon$ is partial (Fact 2.1), we get $\xi_1^-\epsilon/s_1[\sigma]s_2[\sigma]$ (Fact 2.4). As $\xi_1^-\epsilon = (\xi_1\epsilon)^-$ (Fact 5.13) and $\xi_1\epsilon$ is partial, we also know that $(\xi_1\epsilon)^-$ is partial and consequently $(\xi_1\epsilon)^- = (\xi_1\epsilon)^-\eta$ (Fact 2.1). With $\xi_2 = \epsilon$ and $\xi = \xi_1\xi_2$ we have $\xi^-\eta/(s_1s_2)[\sigma]$.
   (c) $\xi_2 = \#$. By the inductive hypothesis for s_1 we know $\xi_1^-\eta/s_1[\sigma]$. In addition we can show that $\#/s_2[\sigma]$ from $\#/s_2$ using Lemma 3.8. Consequently $\xi_1^-\eta\#/s_1[\sigma]s_2[\sigma]$. We know that $\xi_1^-\eta\# = \xi_1^-\eta = (\xi_1\#)^-\eta = (\xi_1\xi_2)^-\eta = \xi^-\eta$ by Fact 2.1 and our assumptions. Consequently $\xi^-\eta/(s_1s_2)[\sigma]$. ∎

There are two kinds of execution paths for a tail-recursive program s under a substitution σ with $\mathcal{D}\sigma \subseteq V$:

1. execution paths of s where no variable that will be substituted appears.
2. execution paths of s that would end with a substituted variable and continue with the execution of the substituted program instead.

Lemma 5.19 Let $\mathsf{trec}_V\ s$ and $\xi/s[\sigma]$ and $\mathcal{D}\sigma \subseteq V$. Then one of the following holds:

1. ξ/s and $\mathcal{V}\xi \parallel \mathcal{D}\sigma$.
2. $\xi = \xi_1^-\xi_2$ and ξ_1/s and ξ_1 ends with variable $x \in \mathcal{D}\sigma$ and $\xi_2/\sigma x$ for some ξ_1, ξ_2.

Proof Induction on $\xi/s[\sigma]$. All cases are easy except those for variables and sequential composition.

- $s = x$ and $\xi/\sigma x$. We make a case distinction on $x \in \mathcal{D}\sigma$.
  1. $x \in \mathcal{D}\sigma$. Then $x\#/x$ and $(x\#)^-\xi = \#\xi = \xi$ (Fact 2.1). As $x\#$ ends with $x \in \mathcal{D}\sigma$ the second case is matched.
  2. $x \notin \mathcal{D}\sigma$. Then ξ/x and by Fact 2.8 we can conclude that $\mathcal{V}\xi \parallel \mathcal{D}\sigma$.
- $s = s_1s_2$ and $\xi = \xi_1\xi_2$ with $\xi_1/s_1[\sigma]$ and $\xi_2/s_2[\sigma]$. By inductive hypothesis for s_1 there are two cases for ξ_1.
  1. ξ_1/s_1 and $\mathcal{V}\xi_1 \parallel \mathcal{D}\sigma$. We distinguish the possible forms of ξ_2 by the inductive hypothesis for s_2:
     (a) ξ_2/s_2 and $\mathcal{V}\xi_2 \parallel \mathcal{D}\sigma$. Then also $\xi_1\xi_2/s_1s_2$ and $\mathcal{V}(\xi_1\xi_2) \parallel \mathcal{D}\sigma$.
     (b) $\xi_2 = \xi_A^-\xi_B$, ξ_A/s_2, ξ_A ends with a variable $x \in \mathcal{D}\sigma$ and $\xi_B/\sigma x$ for some ξ_A, ξ_B. We distinguish whether ξ_1 is partial.
        i. ξ_1 is partial. Then $\xi = \xi_1\xi_2 = \xi_1$ (Fact 2.1) and ξ_1/s_1s_2 by Fact 2.4. Consequently the first case is matched.
        ii. ξ_2 is total. Then $\xi_1\xi_A/s_1s_2$. As ξ_1 is a trace of s_1 and $\mathcal{V}s \parallel V$, we know that $\mathcal{V}\xi_1 \parallel V$ (Fact 2.8) and with $\mathcal{D}\sigma \subseteq V$ we get that $x \notin \mathcal{V}\xi_1$. We conclude with Fact 5.9 that x is terminal in $\xi_1\xi_A$. As ξ_A contains at least one element (namely x), we have $\xi_1\xi_A^- = (\xi_1\xi_A)^-$ (Fact 5.13). Consequently we found a correct decomposition with $\xi = \xi_1\xi_2 = \xi_1\xi_A^-\xi_B = (\xi_1\xi_A)^-\xi_B$ and the second case is matched.
  2. $\xi_1 = \xi_A^-\xi_B$, ξ_A/s_1, ξ_A ends with a variable $x \in \mathcal{D}\sigma$ and $\xi_B/\sigma x$ for some ξ_A, ξ_B. As s_1s_2 is V-tail-recursive, we know $\mathcal{V}s_1 \parallel V$. By Fact 2.8 we have also $\mathcal{V}\xi_A \parallel V$. Together with $\mathcal{D}\sigma \subseteq V$ this contradicts that ξ_A ends with a variable in $\mathcal{D}\sigma$. ∎

5.3.2 Linearizer Correctness

We show the correctness of the linearizer by relating the traces of $L_V\ s\, u$ with the traces of s and u.

The general idea of the linearizer is that it only appends the continuation u to execution paths of s not ending with a variable in V. As V collects the bound variables of s, program paths ending with variables in V are actually not completed, but would loop back.

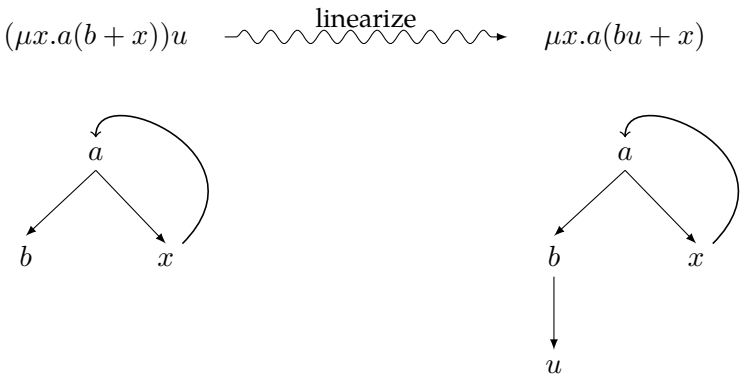

Figure 5.1: Visualization of the change in control flow of the recursion during linearization for the program $(\mu x.a(b+x))u$

There are three possibilities to obtain the trace of $L_V\, s\, u$ from the traces of s and u.

1. If an execution path of s ends with a variable in V, the linearizer has not appended the continuation u to it. Consequently this path is also an execution path of $L_V\, s\, u$.
2. If an execution path of s is partial, it has not already reached a point where the continuation u might have been appended. Consequently it is also a partial execution path of $L_V\, s\, u$.
3. If an execution path of s is total and does not end with a variable in V, the linearizer did append the continuation u to it. Consequently the execution path continues with the execution of u.

Lemma 5.20 Let $\mathsf{trec}_V\ s$. Then:

Claim 1 If ξ/s and ξ ends with $x \in V$, then $\xi/L_V\ s\ u$.

Claim 2 If $\xi /\!\!/ s$ and $\mathcal{V}\xi \parallel V$, then $\xi /\!\!/ L_V\ s\ u$.

Claim 3 If $\xi_1 /\!\!/\!/ s$, $\mathcal{V}\xi_1 \parallel V$ and ξ_2/u, then $\xi_1\xi_2/L_V\ s\ u$.

Proof Induction on $\mathsf{trec}_V\ s$. We show the proofs of Claims 1 to 3 separately.

Proof of Claim 1:

Assume ξ/s and ξ ends with $x \in V$. All cases except for the cases for variables, sequential composition and recursion are easy.

- $s = x$. Case analysis on $x \in V$.
  1. $x \in V$. Then $L_V\ x\ u = x$. Consequently every trace of x is a trace of $L_V\ x\ u$.
  2. $x \notin V$. As $\mathcal{T}x = \{\epsilon, x\epsilon, x\#\}$, ξ cannot end with a variable in V.
- $s = s_1 s_2$. As $\xi/s_1 s_2$, ξ is composed of a trace ξ_1 of s_1 and a trace ξ_2 of s_2. As s_1 does not contain any variables in V (for $s_1 s_2$ being V-tail-recursive), this carries over to ξ_1 (Fact 2.8). We can conclude that for $\xi_1\xi_2$ being a trace that ends with a variable $x \in V$, ξ_1 needs to be a total trace and ξ_2 needs to end with x (Fact 5.11). With the inductive hypothesis for s_2, we know that $\xi_2/L_V\ s_2\ u$. With Claim 3, we get that $\xi_1\xi_2/L_V\ s_1\ (L_V\ s_2\ u)$.
- $s = \mu y.s$. Then $L_V\ (\mu y.s)\ u = \mu y.L_{V\cup\{y\}}\ s\ u$. We use the characterization of recursion with the unfolding operator (Lemma 3.17) and assume $\xi/[s]^n_y$ for some n. We prove by induction on $n \in \mathbb{N}$.
  - $n = 0$. Then $\xi/\varnothing$ and consequently $\xi = \epsilon$ (Fact 2.6). We conclude that $\xi/(\mu y.L_{V\cup\{y\}}\ s\ u)$ by Fact 2.7.
  - $n > 0$. Then $\xi/s^y_{[s]^{n-1}_y}$. We use Lemma 5.19 to distinguish the possible forms of ξ.
    1. ξ/s and $y \notin \mathcal{V}\xi$. Then x needs to be different from y as otherwise ξ could not end with x. By the inductive hypothesis for s, we conclude $\xi/L_{V\cup\{y\}}\ s\ u$. With Lemma 5.17 we get that $\xi/(L_{V\cup\{y\}}\ s\ u)^y_{\mu y.L_{V\cup\{y\}}\ s\ u}$ and consequently also $\xi/\mu y.L_{V\cup\{y\}}\ s\ u$.
    2. $\xi = \xi_1^-\xi_2$ and ξ_1/s, ξ_1 ends with y and $\xi_2/[s]^{n-1}_y$ for some ξ_1, ξ_2. Then ξ_2 ends with x as ξ_1^- cannot contain any variables in $V \cup \{y\}$ (Fact 5.16). As $y \in V \cup \{y\}$ by the inductive hypothesis for s, we can conclude that $\xi_1/L_{V\cup\{y\}}\ s\ u$. In addition the inductive hypothesis for $n-1$ yields for $\xi_2/[s]^{n-1}_y$ that $\xi_2/\mu y.L_{V\cup\{y\}}\ s\ u$. With Lemma 5.18, we get that $\xi_1^-\xi_2/(L_{V\cup\{y\}}\ s\ u)^y_{\mu y.L_{V\cup\{y\}}\ s\ u}$ and consequently $\xi/\mu y.L_{V\cup\{y\}}\ s\ u$.

Proof of Claim 2:

Assume $\xi /\!\!/ s$ and $\mathcal{V}\xi \parallel V$. All cases except for those for variables, sequential composition and recursion are easy.

- $s = x$. We make a case distinction on $x \in V$.
  1. $x \in V$. Then $L_V\, x\, u = x$. Consequently every partial trace of x is also a partial trace of $L_V\, x\, u$.
  2. $x \notin V$. Then $L_V\, x\, u = xu$. By Fact 2.4, we can conclude that every partial trace of x is also a partial trace of $L_V\, x\, u$.
- $s = s_1 s_2$. As $\xi / s_1 s_2$, ξ is composed of a trace ξ_1 of s_1 and a trace ξ_2 of s_2. We distinguish whether ξ_1 is partial.
  1. ξ_1 is partial. Then $\xi = \xi_1 \xi_2 = \xi_1$ (Fact 2.1). Additionally we know from $\mathcal{V}(\xi_1\xi_2) \parallel V$ together with Fact 2.2 that $\mathcal{V}\xi_1 \parallel V$. With the inductive hypothesis for s_1, we get $\xi_1 /\!\!/ L_V\, s_1\, (L_V\, s_2\, u)$.
  2. ξ_1 is total. Then ξ_2 must be partial and by Fact 2.2 we know that $\mathcal{V}\xi_1 \parallel V$ and $\mathcal{V}\xi_2 \parallel V$. Therefore we conclude with the inductive hypothesis for s_2 that $\xi_2 /\!\!/ L_V\, s_2\, u$. With Claim 3 we finally get $\xi_1\xi_2 /\!\!/ L_V\, s_1\, (L_V\, s_2\, u)$
- $s = \mu x.s$. Then $L_V\, (\mu x.s)\, u = \mu x.L_{V\cup\{x\}}\, s\, u$. We use the unfolding-characterization of recursion (Lemma 3.17) and assume $[s]_x^n$ for some $n \in \mathbb{N}$. Induction on $n \in \mathbb{N}$.
  - $n = 0$. Then $\xi / \varnothing$ and consequently $\xi = \epsilon$ (Fact 2.6). We conclude $\xi / \mu x.L_{V\cup\{x\}}\, s\, u$ (Fact 2.7).
  - $n > 0$. Then $\xi / s^x_{[s]_x^{n-1}}$. We use Lemma 5.19 to distinguish the possible forms of ξ.
    1. $\xi /\!\!/ s$ and $x \notin \mathcal{V}\xi$. Then also $\mathcal{V}\xi \parallel V \cup \{x\}$ (by $\mathcal{V}\xi \parallel V$). By the inductive hypothesis for s we conclude that $\xi / L_{V\cup\{x\}}\, s\, u$. As $x \notin \mathcal{V}\xi$ we get $\xi /\!\!/ (L_{V\cup\{x\}}\, s\, u)^x_{\mu x.L_{V\cup\{x\}}\, s\, u}$ by Lemma 5.17 and consequently also $\xi /\!\!/ \mu x.L_{V\cup\{x\}}\, s\, u$.
    2. $\xi = \xi_1^- \xi_2$ and ξ_1 / s, ξ_1 ends with x and $\xi_2 / [s]_x^{n-1}$ for some ξ_1, ξ_2. We distinguish whether ξ_1 is partial.
       (a) ξ_1 is partial. Then ξ_1^- is partial and consequently $\xi = \xi_1^- \xi_2 = \xi_1^-$. By Claim 1, we get from ξ_1 / s that $\xi_1 /\!\!/ L_{V\cup x}\, s\, u$. From Fact 5.15, we know $\xi_1^- /\!\!/ L_{V\cup x}\, s\, u$. Additionally we know by Fact 5.14 that $x \notin \mathcal{V}\xi_1^-$ and consequently we conclude with Lemma 5.17 $\xi_1^- / (L_{V\cup\{x\}}\, s\, u)^x_{\mu x.L_{V\cup\{x\}}\, s\, u}$ and therefore also $\xi /\!\!/ \mu x.L_{V\cup\{x\}}\, s\, u$.
       (b) ξ_1 is total. Then ξ_2 needs to be partial and $\mathcal{V}\xi_2 \parallel V$ (Fact 2.2). By the inductive hypothesis for $n-1$ we conclude $\xi_2 /\!\!/ \mu x.L_{V\cup\{x\}}\, s\, u$. By Claim 1 we get from ξ_1 / s that $\xi_1 / L_{V\cup\{x\}}\, s\, u$. By Lemma 5.18 we have

$$\xi_1^- \xi_2 /\!\!/ (L_{V\cup\{x\}}\ s\ u)^x_{\mu x.L_{V\cup\{x\}}\ s\ u} \text{ and consequently } \xi /\!\!/ \mu x.L_{V\cup\{x\}}\ s\ u.$$

Proof of Claim 3:

Assume $\xi_1 /\!\!/\!/ s$, $\mathcal{V}\xi_1 disjoint V$ and ξ_2 / u. All cases are easy except for those for variables and recursion.

- $s = x$. Then $\xi_1 = x\#$. By $\mathcal{V}\xi_1 \parallel V$, we conclude that $x \notin V$. Consequently $L_V\ x\ u = xu$. From $\xi_1 /\!\!/\!/ x$ and ξ_2 / u, we get that $\xi_1\xi_2 / L_V\ x\ u$.
- $s = \mu x.s$. Then $L_V\ (\mu x.s)\ u = \mu x.L_{V\cup\{x\}}\ s\ u$. We use the unfolding characterization of recursion (Lemma 3.17) and assume $\xi_1 /\!\!/\!/ [s]^n_x$ for some $n \in \mathbb{N}$. Induction on $n \in \mathbb{N}$.
  - $n = 0$. Then $\xi_1 /\!\!/\!/ \varnothing$. This is directly contradictory as $\varnothing$ has only partial traces (Fact 2.6).
  - $n > 0$. Then $\xi_1 /\!\!/\!/ s^x_{[s]^{n-1}_x}$. We deconstruct ξ_1 by Lemma 5.19.
    1. $\xi_1 /\!\!/\!/ s$ and $x \notin \mathcal{V}\xi_1$. Then $\mathcal{V}\xi_1 \parallel V \cup \{x\}$. By the inductive hypothesis for s we get $\xi_1\xi_2 / L_{V\cup\{x\}}\ s\ u$. As $x \notin \mathcal{V}u$ and consequently $x \notin \mathcal{V}\xi_2$ (Fact 2.8), we conclude that $\xi_1\xi_2 / (L_{V\cup\{x\}}\ s\ u)^x_{\mu x.L_{V\cup\{x\}}\ s\ u}$ by Lemma 5.17. By definition, we get $\xi_1\xi_2 / \mu x.L_{V\cup\{x\}}\ s\ u$.
    2. $\xi_1 = \xi_A^- \xi_B$ and ξ_A ends with x and $\xi_A /\!\!/\!/ s$ and $\xi_B /\!\!/\!/ [s]^{n-1}_x$. With the inductive hypothesis for $n - 1$ we get $\xi_B\xi_2 / \mu x.L_{V\cup\{x\}}\ s\ u$. With Claim 1, $\xi_A /\!\!/\!/ s$ gives us $\xi_A / L_{V\cup\{x\}}\ s\ u$. By Lemma 5.18 we have $\xi_A^-(\xi_B\xi_2) / (L_{V\cup\{x\}}\ s\ u)^x_{\mu x.L_{V\cup\{x\}}\ s\ u}$ and consequently also $\xi_1\xi_2 / \mu x.L_{V\cup\{x\}}\ s\ u$. ∎

After studying how the traces of $L_V\ s\ u$ can be composed from traces of s and u, we will show that we can tie the traces of $L_V\ s\ u$ to the traces of s and u accordingly.

Execution paths of $L_V\ s\ u$ can be of three forms:

1. paths of s which never reached the end of s
2. paths that ended with a variable in V
3. paths that passed through s completely without appearances of variables in V and continued with the execution of u

Lemma 5.21 Let $\mathsf{trec}_V\ s$, $\mathsf{trec}_V\ u$ and $\xi / L_V\ s\ u$. Then one of the following:

1. $\xi /\!\!/ s$ and $\mathcal{V}\xi \parallel V$.
2. ξ / s and ξ ends with a variable $x \in V$.
3. $\xi = \xi_1\xi_2$ and $\xi_1 /\!\!/\!/ s$ and $\mathcal{V}\xi_1 \parallel V$ and ξ_2 / u for some ξ_1, ξ_2.

Proof Induction on $\mathsf{trec}\ V\ s$. Assume $\mathsf{trec}_V\ s$ and $\mathsf{trec}_V\ u$. We consider the case $\xi = \epsilon$ once. Traces of the form ϵ match the first case as $\epsilon /\!\!/ s$ and $\mathcal{V}\epsilon = \emptyset$.

- $s = 1$. Then $L_V\ 1\ u = u$. The third case matches as a trace ξ of u can be decomposed to $\#\xi$ and $\#/1$ and $\mathcal{V}\# = \emptyset$.

- $s = x$. Case distinction on $x \in V$.
  1. $x \in V$. Then $L_V\, x\, u = x$. Then the second case matches trivially.
  2. $x \notin V$. Then $L_V\, x\, u = xu$. Then a trace ξ of xu can be decomposed into a trace ξ_1 of x and a trace ξ_2 of u. We distinguish whether ξ_1 is partial.
     (a) ξ_1 is partial. Then $\xi = \xi_1\xi_2 = \xi_1$ (Fact 2.1). Consequently $\xi /\!\!/ x$ and $\mathcal{V}\xi \parallel V$ as $x \notin V$.
     (b) ξ_1 is total. Then the third case matches. $\mathcal{V}\xi_1 \parallel V$ holds as $x \notin V$.
- $s = s_1 s_2$. Then $L_V\, (s_1 s_2)\, u = L_V\, s_1\, (L_V\, s_2\, u)$. By inductive hypothesis for s_1 we distinguish three cases for traces ξ of $L_V\, s_1\, (L_V\, s_2\, u)$.
  1. $\xi /\!\!/ s_1$ and $\mathcal{V}\xi \parallel V$. Then also $\xi /\!\!/ s_1 s_2$ (Fact 2.4) and the first case matches.
  2. ξ / s_1 and ξ ends with $x \in V$. As $\mathcal{V}s_1 \parallel V$ (for $s_1 s_2$ being V-tail-recursive), we know that $\mathcal{V}\xi \parallel V$ (Fact 2.8). Consequently, this case is contradictory.
  3. $\xi = \xi_1\xi_2$ and $\xi_1 /\!\!/\!/ s_1$ and $\mathcal{V}\xi_1 \parallel V$ and $\xi_2 / L_V\, s_2\, u$ for some ξ_1, ξ_2. We make a distinction on the form of ξ_2 using the inductive hypothesis for s_2.
     (a) $\xi_2 /\!\!/ s_2$ and $\mathcal{V}\xi_2 \parallel V$. Then $\xi_1\xi_2$ is a partial trace of $s_1 s_2$ and $\mathcal{V}(\xi_1\xi_2) \parallel V$ as $\mathcal{V}\xi_1 \parallel V$ and $\mathcal{V}\xi_2 \parallel V$ (Fact 2.2). Consequently the first case matches.
     (b) ξ_2 / s_2 and ξ_2 ends with $x \in V$. Then $\xi_1\xi_2$ is a trace of $s_1 s_2$ ending with x as ξ_1 is total and does not contain any variables in V. Consequently the second case matches.
     (c) $\xi_2 = \xi_A\xi_B$ and $\xi_A /\!\!/\!/ s_2$ and $\mathcal{V}\xi_A \parallel V$ and ξ_B / u for some ξ_A, ξ_B. Then $\xi = (\xi_1\xi_A)\xi_B$ and $\xi_1\xi_A$ is a total trace of $s_1 s_2$. Additionally $\mathcal{V}(\xi_1\xi_A) \parallel V$ holds as $\mathcal{V}\xi_1 \parallel V$ and $\mathcal{V}\xi_A \parallel V$ (Fact 2.2). Consequently the third case matches.
- $s = s_1 + s_2$. Then $L_V\, (s_1 + s_2)\, u = L_V\, s_1\, u + L_V\, s_2\, u$. Consequently either $\xi / L_V\, s_2\, u$ or $\xi / L_V\, s_1\, u$. We show the claim for $\xi / L_V\, s_2\, u$, the other case works analogously using the inductive hypothesis for s_2. We distinguish the form of ξ using the inductive hypothesis for s_1:
  1. $\xi /\!\!/ s_1$ and $\mathcal{V}\xi \parallel V$. Then by definition $\xi /\!\!/ s_1 + s_2$ and the first case matches.
  2. ξ / s_1 and ξ ends with $x \in V$. Then also $\xi / s + t$ and the second case matches.
  3. $\xi = \xi_1\xi_2$, $\xi_1 /\!\!/\!/ s_1$, $\mathcal{V}\xi_1 \parallel V$ and ξ_2 / u for some ξ_1, ξ_2. Then also $\xi_1 /\!\!/\!/ s + t$ and the third case matches.
- $s = \mu x.s$. Then $L_V\, (\mu x.s)\, u = \mu x.L_{V\cup\{x\}}\, s\, u$. We use the unfolding characterization of recursion and assume $\xi / [L_{V\cup\{x\}}\, s\, u]^n_x$ for some $n \in \mathbb{N}$. Induction on $n \in \mathbb{N}$.
  - $n = 0$. Then $\xi / \varnothing$ and consequently $\xi = \epsilon$ (Fact 2.6). Consequently, the first case matches.
  - $n > 0$. Then $\xi / (L_{V\cup\{x\}}\, s\, u)^x_{[L_{V\cup\{x\}}\, s\, u]^{n-1}_x}$. As u does not contain x, we can

strengthen $\mathsf{trec}_V\ u$ to $\mathsf{trec}_{V\cup\{x\}}\ u$ (Fact 4.1). By Lemma 5.7 and $\mathsf{trec}_{V\cup\{x\}}\ s$ we know that $\mathsf{trec}_{V\cup\{x\}}\ (L_{V\cup\{x\}}\ s\ u)$. This allows us to distinguish the different forms of s using Lemma 5.19.

1. $\xi / L_{V\cup\{x\}}\ s\ u$ and $x \notin \mathcal{V}\xi$. By the inductive hypothesis for s, we can again distinguish different forms of ξ.
   (a) $\xi /\!\!/ s$ and $\mathcal{V}\xi \parallel V \cup \{x\}$. Then also $\xi /\!\!/ s^x_{\mu x.s}$ by Lemma 5.17 and consequently $\xi /\!\!/ \mu x.s$ and $\mathcal{V}\xi \parallel V$.
   (b) ξ / s and ξ ends with $y \in V\cup\{x\}$. As $x \notin \mathcal{V}\xi$, we know that $y \in V\backslash\{x\}$. As $x \notin \mathcal{V}\xi$, we conclude by Lemma 5.17 that $\xi / s^x_{\mu x.s}$ and consequently $\xi / \mu x.s$. As ξ ends with a variable $y \in V\backslash\{x\}$, the second case matches.
   (c) $\xi = \xi_1\xi_2$ and $\xi_1 /\!\!/\!/ s$ and $\mathcal{V}\xi_1 \parallel V \cup \{x\}$ and ξ_2 / u for some ξ_1, ξ_2. As $x \notin \mathcal{V}\xi_1$, we get $\xi_1 /\!\!/\!/ s^x_{\mu x.s}$ by Lemma 5.17. Consequently, $\xi_1 /\!\!/\!/ \mu x.s$ and the third case matches.
2. $\xi = \xi_1^- \xi_2$ and $\xi_1 / L_{V\cup\{x\}}\ s\ u$ and ξ_1 ends with x and $\xi_2 / [L_{V\cup\{x\}}\ s\ u]^{n-1}_x$ for some ξ_1, ξ_2. By the inductive hypothesis for s, we distinguish different forms of ξ_1.
   (a) $\xi_1 /\!\!/ s$ and $\mathcal{V}\xi_1 \parallel V \cup \{x\}$. Consequently $x \notin \mathcal{V}\xi_1$ what directly contradicts that ξ_1 ends with x.
   (b) ξ_1 / s and ξ_1 ends with $y \in V \cup \{x\}$. We distinguish whether ξ_1 is partial
      i. ξ_1 is partial. Then ξ_1^- is also partial and $\xi = \xi_1^-\xi_2 = \xi_1^-$ (Fact 2.1). By the prefix-closedness (Fact 2.10), we conclude that $\xi_1^- /\!\!/ s$. As s is $(V\cup\{x\})$-tail-recursive, we know that ξ_1 is $(V\cup\{x\})$-tail-recursive as well (Fact 5.12). Consequently $\mathcal{V}\xi_1^- \parallel V \cup \{x\}$ (Fact 5.16). As $x \notin \mathcal{V}\xi_1^-$, we conclude $\xi_1^- /\!\!/ s^x_{\mu x.s}$ by Lemma 5.17. Consequently we have $\xi /\!\!/ \mu x.s$ and $\mathcal{V}\xi \parallel V$.
      ii. ξ_1 is total. Then ξ_1^- is also total. As s is $(V \cup \{x\})$-tail-recursive, we know that ξ_1 is $(V \cup \{x\})$-tail-recursive, too (Fact 5.12). Consequently $\mathcal{V}\xi_1^- \parallel V\cup\{x\}$ (Fact 5.16). We use the inductive hypothesis for $n-1$ to argue about the form of ξ_2.
         A. $\xi_2 /\!\!/ \mu x.s$ and $\mathcal{V}\xi_2 \parallel V$. With Lemma 5.18, we get $\xi_1^-\xi_2 /\!\!/ s^x_{\mu x.s}$ and consequently $\xi /\!\!/ \mu x.s$. As $\mathcal{V}\xi_1^- \parallel V \cup \{x\}$ and $\mathcal{V}\xi_2 \parallel V$, we also know that $\mathcal{V}\xi \parallel V$ (Fact 2.2) and the first case matches.
         B. $\xi_2 / \mu x.s$ and ξ_2 ends with $y \in V$. With Lemma 5.18, we get $\xi_1^-\xi_2 /\!\!/ s^x_{\mu x.s}$ and consequently $\xi /\!\!/ \mu x.s$. As $x \notin \mathcal{V}\xi_1^-$, ξ_1^- is total, and ξ_2 ends with $y \in V$, we know that ξ also ends with $y \in V$ (Fact 5.9) and the second case matches.
         C. $\xi_2 = \xi_A\xi_B$ and $\xi_A /\!\!/\!/ \mu x.s$ and $\mathcal{V}\xi_A \parallel V$ and ξ_B / u. Then

by Lemma 5.18 we have $\xi_1^- \xi_A /\!\!/\!/ s^x_{\mu x.s}$ and consequently $\xi_1^- \xi_A /\!\!/\!/ \mu x.s$. As $\xi = \xi_1^-(\xi_A \xi_B) = (\xi_1^- \xi_A)\xi_B$, the third case matches.

(c) $\xi_1 = \xi_A \xi_B$ and $\xi_A /\!\!/\!/ s$ and $\mathcal{V}\xi_A \parallel V \cup \{x\}$ and ξ_B / u. As ξ_1 ends with x and ξ_A is total and does not contain x, ξ_B needs to end with x (Fact 5.10). As u does not contain x as free variable by definition of the linearizer, ξ_B cannot contain x as well (Fact 2.8). This contradicts that ξ_B ends with x. ∎

Theorem 5.22 Let $\mathsf{trec}_\emptyset\, s$ and $\mathsf{ltrec}_\emptyset\, u$, then $\mathsf{ltrec}_V\, (L_\emptyset\, s\, u)$ and $su \approx L_\emptyset\, s\, u$.

Proof Assume $\mathsf{trec}_\emptyset\, s$ and $\mathsf{ltrec}_\emptyset\, u$. Then $\mathsf{ltrec}_V\, (L_\emptyset\, s\, u)$ follows directly from Lemma 5.8. To show that $su \approx L_\emptyset\, s\, u$, we consider both directions of the equivalence separately.

- Assume ξ / su. Then ξ is composed of a trace ξ_1 of s and a trace ξ_2 of u. Trivially $\mathcal{V}\xi_1 \parallel \emptyset$ holds. We distinguish whether ξ_1 is partial.
  1. ξ_1 is partial. Then $\xi = \xi_1\xi_2 = \xi_1$ and ξ_1 is a trace of $L_\emptyset\, s\, u$ by Claim 2 of Lemma 5.20.
  2. ξ_1 is total. Then $\xi_1\xi_2$ is a trace of $L_\emptyset\, s\, u$ by Claim 3 of Lemma 5.20.
- Assume $\xi / L_\emptyset\, s\, u$. Then by Lemma 5.21 there are three possibilites for ξ:
  1. $\xi /\!\!/ s$. Then also ξ / su by Fact 2.4.
  2. ξ / s and x ends with $x \in \emptyset$. This is directly contradictory.
  3. $\xi = \xi_1\xi_2$, $\xi_1 /\!\!/\!/ s$ and ξ_2 / u for some $\xi_1\ \xi_2$. Then also $\xi_1\xi_2 / su$ by definition. ∎

Chapter 6

Context-Free Programs and abstract IMP

To study the correspondence between context-free programs and imperative programming languages, we consider the abstract idealized programming language *abstract IMP* with a big-step semantics. Abstract IMP abstracts from variable assignments and boolean conditions by considering them as relations on the memory state. This abstraction allows us to show a translation from abstract IMP programs to context-free programs with actions and tests as free variables. We call the context-free program resulting from this translation the *context-free abstraction* of the abstract IMP program. We show how the big-step semantics of an abstract IMP program can be reconstructed from the trace semantics of its context-free abstraction.

The syntax of abstract IMP is obtained by the following grammar:

$$c, d \ ::= \ \mathsf{skip} \mid a \mid c; d \mid \mathsf{if}\ b\ c\ d \mid \mathsf{while}\ b\ c \qquad (a \in \mathcal{A}) \quad (b \in \mathcal{B})$$

In contrast to Winskel's IMP [13], we assume abstract actions a instead of variable assignments. We denote the set of actions with $\mathcal{A}$. In addition we assume a subset of actions $\mathcal{B}$ that is closed under negation. We refer to the elements $b \in \mathcal{B}$ as *tests*.

In order to define a state based semantics for IMP programs, we assume a set Σ of abstract states. To describe the effect of actions and tests, we assume a predicate $\mathsf{exec} : \Sigma \times \mathcal{A} \times \Sigma$. The predicate exec relates a state σ and an action a with a state τ if the execution of a in σ may result in τ.

Tests $b \in \mathcal{B}$ can be seen as partial identities on states. The predicate exec either relates a state σ and a test b with the same state σ or with no state at all. If σ and b are not related with another state, then σ and the negation $\overline{b}$ are related with σ. Accordingly, if σ and $\overline{b}$ are related with σ, then σ and b are not.

We obtain the following axiomatization of actions and tests:

1. $\mathcal{A}$: set of of abstract actions
2. $\mathcal{B} \subseteq \mathcal{A}$: set of tests
3. $\overline{b} : \mathcal{B} \to \mathcal{B}$
4. $\overline{b} \in \mathcal{B} \leftrightarrow b \in \mathcal{B}$
5. $\mathsf{exec} : \Sigma \times \mathcal{A} \times \Sigma$
6. $\mathsf{exec}\ \sigma\ b\ \tau \to \sigma = \tau$
7. $\mathsf{exec}\ \sigma\ \overline{b}\ \sigma \leftrightarrow \neg\mathsf{exec}\ \sigma\ b\ \sigma$

We give a big-step semantics for abstract IMP.

$$\frac{}{(\sigma, \mathsf{skip}) \Rightarrow \sigma} \qquad \frac{\mathsf{exec}\ \sigma\ a\ \tau}{(\sigma, a) \Rightarrow \tau} \qquad \frac{\mathsf{exec}\ \sigma\ b\ \sigma \qquad (\sigma, c) \Rightarrow \tau}{(\sigma, \mathsf{if}\ b\ c\ d) \Rightarrow \tau}$$

$$\frac{\neg\mathsf{exec}\ \sigma\ b\ \sigma \qquad (\sigma, d) \Rightarrow \tau}{(\sigma, \mathsf{if}\ b\ c\ d) \Rightarrow \tau} \qquad \frac{\mathsf{exec}\ \sigma\ b\ \sigma \qquad (\sigma, c) \Rightarrow \sigma' \qquad (\sigma', \mathsf{while}\ b\ c) \Rightarrow \tau}{(\sigma, \mathsf{while}\ b\ c) \Rightarrow \tau}$$

$$\frac{\neg\mathsf{exec}\ \sigma\ b\ \sigma}{(\sigma, \mathsf{while}\ b\ c) \Rightarrow \sigma}$$

Two abstract IMP program c and d are called equivalent if the big-step semantics yields the same final states for all initial states for c and d.

$$c \approx_{\text{IMP}} d := \forall \sigma \tau.\ (\sigma, c) \Rightarrow \tau \leftrightarrow (\sigma, d) \Rightarrow \tau$$

The abstraction from variable assignments and conditionals in abstract IMP, allows us to encode abstract IMP programs as context-free programs using the following encodings:

$$\mathsf{if}\ b\ c\ d \rightsquigarrow bs + \overline{b}t$$
$$\mathsf{while}\ b\ c \rightsquigarrow \mu x.\overline{b} + bsx$$

We call the context-free program resulting from the encoding of an IMP program c the *context-free abstraction* of c.

We define a function $\mathsf{cfa} : \mathsf{imp} \to \mathsf{cfp}$ that constructs the context-free abstraction for an abstract IMP program. Where imp denotes the set of all abstract IMP programs.

$$
\begin{aligned}
\mathsf{cfa} &: \mathsf{imp} \to \mathsf{cfp} \\
\mathsf{cfa}\ \mathsf{skip} &:= 1 \\
\mathsf{cfa}\ a &:= a \\
\mathsf{cfa}\ (c; d) &:= (\mathsf{cfa}\ c)(\mathsf{cfa}\ d) \\
\mathsf{cfa}\ (\mathsf{if}\ b\ c\ d) &:= b(\mathsf{cfa}\ c) + \overline{b}(\mathsf{cfa}\ d) \\
\mathsf{cfa}\ (\mathsf{while}\ b\ c) &:= \mu x.b(\mathsf{cfa}\ c)x + \overline{b}1 \qquad b, \overline{b} \neq x
\end{aligned}
$$

Remark. Semantically equivalent formulations for the context-free abstraction for while are $\mu x.b(\mathsf{cfa}\ c)x + \overline{b}$ or $\mu x.\overline{b} + b(\mathsf{cfa}\ c)x$. For the proofs the chosen formulation is useful as it fulfils the following unfolding law:

$$(b(\mathsf{cfa}\ c)x + \overline{b}1)^x_{\mu x.b(\mathsf{cfa}\ c)x + \overline{b}1} = \mathsf{cfa}\ (\mathsf{if}\ b\ (c; \mathsf{while}\ b\ c)\ \mathsf{skip})$$

Fact 6.1 cfa c is tail-recursive.

We will show that the equivalence of the context-free abstractions of two abstract IMP programs is a sufficient condition for their equivalence with respect to the big-step semantics:

$$\mathsf{cfa}\ c \approx \mathsf{cfa}\ d \to c \approx_{\mathrm{IMP}} d$$

To relate the trace semantics of context-free abstractions with the big-step semantics of abstract IMP programs, we develop a notion for the execution of a trace of actions on a state.

We define a predicate run $\sigma\ \xi\ \tau$ that satisfies the following equivalence:

$$\mathsf{run}\ \sigma\ \xi\ \tau \leftrightarrow \xi = a_1, \ldots a_n\# \wedge \exists \sigma_0, \ldots \sigma_n. \forall i < n.\ \mathsf{exec}\ \sigma_i\ a_{i+1}\ \sigma_{i+1} \wedge \sigma = \sigma_1 \wedge \tau = \sigma_n$$

The predicate run is inductively defined by the following inference rules:

$$\frac{}{\mathsf{run}\ \sigma\ \#\ \sigma} \qquad\qquad \frac{\mathsf{exec}\ \sigma\ a\ \sigma' \qquad \mathsf{run}\ \sigma'\ \xi\ \tau}{\mathsf{run}\ \sigma\ (a\xi)\ \tau}$$

Intuitively, run $\sigma\ \xi\ \tau$ means that the actions of ξ consecutively invoked on σ result in τ.

Fact 6.2

1. Let run $\sigma\ \xi\ \sigma'$ and run $\sigma'\ \eta\ \tau$. Then run $\sigma\ (\xi\eta)\ \tau$
2. Let run $\sigma\ (\xi\eta)\ \tau$. Then there is a state σ' such that run $\sigma\ \xi\ \sigma'$ and run $\sigma'\ \eta\ \tau$.

Lemma 6.3 Let $(\sigma, c) \Rightarrow \tau$. Then there is a trace ξ such that $\xi/\mathsf{cfa}\ c$ and $\mathsf{run}\ \sigma\ \xi\ \tau$.

Proof By induction on $(\sigma, c) \Rightarrow \tau$. The case for sequential composition uses Fact 6.2. ∎

Lemma 6.4 Let $\xi/\mathsf{cfa}\ c$ and $\mathsf{run}\ \sigma\ \xi\ \tau$. Then $(\sigma, c) \Rightarrow \tau$.

Proof By induction on $\xi/\mathsf{cfa}\ c$. The case for sequential composition uses Fact 6.2. ∎

Theorem 6.5 Let $\mathsf{cfa}\ c \approx \mathsf{cfa}\ d$. Then $c \approx_{\mathrm{IMP}} d$.

Proof Follows with Lemmas 6.3 and 6.4. ∎

Chapter 7

Conclusions

In this thesis, we studied tail-recursive context-free programs with a trace semantics. Context-free programs are an abstract model for imperative programming languages. They consist of uninterpreted actions as atomic programs and abstract control structures known from regular expressions enriched with a recursion operator generalizing Kleene iteration. We focused on programs where recursions are restricted to tail recursions. We showed and verified equivalence transformations for tail-recursive programs to two kinds of normal forms: regular programs and linear tail-recursive programs. The trace semantics used for the verification of our results characterizes each program by the partial and total action sequences it can invoke. Thus, trace semantics allows to obtain meaningful descriptions of terminating as well as non-terminating programs.

7.1 Discussion

Context-free programs do not represent a realistic programming language. They do not allow conditions or guards to influence the control flow, but assume non-determinism whenever different execution paths are possible. The non-determinism is expressed in the trace semantics by recording all possible traces. The traces of a program describe how the program might behave. Nevertheless, context-free programs can be used to describe more realistic imperative programming languages if the abstract actions are interpreted as shown in Chapter 6. For this reason, we state that the transformations and proof techniques used in this thesis can be transferred to many other settings including reactive versions of IMP.

With the translation from tail-recursive programs to regular programs (Chapter 4), we showed the correspondence of tail-recursion and iteration. The translation shows that long-distance recursion can be resolved to short-distance recursion.

With the translation from tail-recursive to linear-programs (Chapter 5) we studied the compilation step needed to translate high level programming languages to linear register transfer languages and showed its correctness in a generalized setting.

We have seen that a compiler from regular programs to linear tail-recursive program is much easier to verify than a direct compiler from tail-recursive programs to tail-recursive linear programs. Since regular programs are already short distance, we can apply equivalences locally to argue about the correctness of the transformation. We also gave a direct linearizer that respects the program structure. The verification of the direct linearizer is much more difficult to verify since it needs to keep track of bound variables. The correctness proof relies on a detailed study of the traces of tail-recursive programs.

7.2 Future Work

An interesting problem not addressed in this thesis is the correspondence of the total trace languages of context-free programs with context-free languages. Winter et al. [14] show several related results for context-free languages using co-algebras. We would like to use another approach that extends context-free programs with mutual recursion. Context-free programs with mutual recursion should allow to describe context-free grammars directly. Accordingly, resolving mutual recursion should correspond to the translation of context-free grammars to context-free programs.

Another problem that is left open is the formal exploration of the relationship between the trace semantics considered here and the small-step semantics for imperative programs. It would be interesting to prove that given a concrete initial state and an interpretation of actions as assignments and tests, the small-step semantics can be reconstructed from the trace semantics.

Another interesting topic is the investigation of Brzozowski derivatives [3] for context-free and tail-recursive context-free programs. Brzozowski derivatives are a means to construct deterministic automata from regular expressions [2]. Additionally, Brzozowski derivatives allow an efficient membership test for regular languages described by regular expressions [8]. We would like to obtain similar results for tail-recursive programs. In addition, derivatives might be a means to show the decidability of program equivalence. Almeida, Broda and Moreira [1] show the decidability of the equivalence of two KAT expressions using partial derivatives.

Appendix A

Coq Formalization

The results of this thesis are carried out using the proof assistant Coq. The formalization is available at www.ps.uni-saarland.de/~schneidewind/bachelor. The Coq files of the formalization were compiled with Coq 8.4pl5.

The Coq formalization uses De Bruijn indices instead of variables. For this reason the formulation of some statements in this thesis may differ from the corresponding formulation in Coq. The main differences can be found in the formulation of the predicates for tail recursion, regularity and linearity as well as in the treatment of bound variables and the related lemmas.

For the realization of De Bruijn indices in Coq, we used the library AutoSubst [10]. In addition we used some helpful tactics defined in the files `Util.v` and `AutoIndTac.v` provided by Steven Schäfer and Sigurd Schneider.

Organization of the Files

Context-free Programs

The syntax of context-free programs is formalized in the file `CFP.v`. In addition, this file contains several technical notions and helpful facts about substitutions into context-free programs.

Traces

Traces are defined in the file `Traces.v`. In addition, this file contains trace concatenation and all properties of traces concerning free variables, partiality and totality, last element removal, and the endings of traces.

Trace Semantics

The trace semantics for context-free programs is defined in the file `TraceSemantics.v`. In addition, the properties of trace semantics stated in Chapter 2 are proven in this file. Furthermore, program equivalence based on traces is introduced and proven to be an equivalence relation.

Free Variables

Facts about the free variables of context-free programs are proven in `FreeVariables.v`. As we assume De Bruijn indices instead of variables in the Coq Development, we introduced a predicate that specifies an interval of indices not occurring freely in a program. Facts about this predicate are in `FreeVariables.v`.

Important Equivalences

The file Equivalences.v contains the equivalences stated in Fact 3.2.

Substitutivity

Substitutivity of program equivalence is proven in the file `Substitutivity.v`. This file contains all definitions and proofs from Section 3.1.

Congruence

In the file `Congruence.v` program equivalence is proven to be a congruence relation. The file contains the proofs of the congruence laws from Section 3.2. The definition of the unfolding operator (Definition 3.13) and the statements showing that the traces of recursions can be characterized using the unfolding operator are located in the file `Unfolding.v`.

Tail recursion

Tail recursion of context-free programs is defined in the file `TailRecursion.v`. In addition, this file contains the properties of the tail recursion predicate stated in Section 4.1.

Regularity

Regularity of context-free programs is defined in the file `Regularity.v`. This file also contains the properties of the two regularity predicates stated in Section 4.2.

Regularizer

The regularizer for tail-recursive programs is defined in the file `Regularizer.v`. This file additionally contains the correctness proofs from Section 4.3.1. The distributivity law for decomposed recursions used for the verification (Section 4.3) is proven in the file `DistributeFix.v`.

Linearity

Linearity of context-free programs is defined in the file `Linearity.v`. In addition, this file contains the properties of the linearity predicate stated in Section 5.1.

Linearizer for regular programs

The linearizer for regular programs is defined in the file `Regularizer.v`. This file contains all correctness proofs stated in Section 5.2.

Linearizer for tail-recursive programs

The linearizer for tail-recursive programs is defined in the file `DirectLinearizer.v`. In addition, this file contains the correctness proofs stated in Section 5.3 as well as the lemmas concerning the traces of tail-recursive programs under substitution (Lemmas 5.17 to 5.19).

Context-free programs and abstract IMP

The encoding of abstract IMP programs using context-free programs is defined in the file `IMP.v`. This file additionally contains all proofs from Chapter 6 concerning the correspondence of the big-step semantics of abstract IMP programs and the trace semantics of their context-free abstractions.

Bibliography

[1] Ricardo Almeida, Sabine Broda, and Nelma Moreira. Deciding KAT and Hoare logic with derivatives. In *Proceedings Third International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2012, Napoli, Italy, September 6-8, 2012.*, pages 127–140, 2012.

[2] Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical computer science*, 48:117–126, 1986.

[3] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[4] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy, and Formal Derivation of Programs. In David Gries, editor, *Programming Methodology*, Texts and Monographs in Computer Science, pages 166–175. Springer New York, 1978.

[5] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of computer and system sciences*, 18(2):194–211, 1979.

[6] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[7] Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[8] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(02):173–190, 2009.

[9] Helmut A. Partsch. *Specification and transformation of programs: a formal approach to software development*. Springer Science & Business Media, 2012.

[10] Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In Xingyuan Zhang and Christian Urban, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015*, LNAI. Springer-Verlag, Aug 2015. To appear.

[11] Sigurd Schneider, Gert Smolka, and Sebastian Hack. A linear first-order functional intermediate language for verified compilers. In Xingyuan Zhang and Christian Urban, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015*, LNAI. Springer-Verlag, 2015.

[12] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 8.4 edition, 2012. `http://coq.inria.fr/`.

[13] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[14] Joost Winter, Marcello M. Bonsangue, and Jan Rutten. *Context-free languages, coalgebraically*. Springer, 2011.