Saarland University

Faculty of Mathematics and Computer Science

Master's Thesis

Mechanized Undecidability of Subtyping in System F

Author
Roberto Álvarez Castro

Advisor
Dr. Yannick Forster

Reviewers
Prof. Dr. Gert Smolka
Dr. Yannick Forster

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig 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, 1th April, 2022

Abstract

Subtyping and universal quantification are two orthogonal features often desired in a type system, and ubiquitous among programming languages. The combination of both these features gives rise to *bounded quantification* which restricts type abstraction so that only subtypes of a given bound can be used to instantiate. As System F formalizes universal quantification, the type system formalizing bounded quantification is called System $F_{\leqslant:}$.

Pierce [33] shows that bounded quantification makes subtyping and type checking undecidable in System $F_{\leqslant:}$. The proof consists of a series of refinements to System $F_{\leqslant:}$ without arrow types, as they are not necessary to show undecidability. The subtyping relation on the refined system is shown to coincide with the full system while being able to encode the transition function of an intermediate machine model; the rowing machines. Finally the halting problem of rowing machines is shown to be many-one reducible from the halting problem of two-counter machines, which is a well-known undecidable problem.

In this work we give a machine-checked presentation of Pierce's proof using the synthetic approach of the Coq Library of Undecidability Proofs [22]. The mechanization is built around syntax produced by Autosubst 2 [40] as the tool provides several useful variants of syntax that significantly simplify the proof. In particular we use *unscoped* and *well-scoped* syntax, alongside with *polyadic* binders, and show how to encode them appropriately using *bounded* extensionality.

We formalize Pierce's proof showing explicitly the necessary inductive arguments in each step, for example complete induction on the height of derivations, the details of which are glossed over in the textbook presentation. Finally, we describe several mechanization artifacts that are unrelated to the mathematical argument but are nevertheless interesting as they simplify the machine-checked proof.

Acknowledgements

This thesis could not have been done without the support, encouragement, and assistance of several people to whom I owe much and cannot offer more here than my heartfelt gratitude.

I am extremely grateful to my advisor Yannick for his invaluable advice, guidance and patience throughout this project. I would like to thank all the members in the Programming Systems Lab, specially to Prof. Dr. Gert Smolka for fostering a welcoming community in which to learn and discuss topics related to type theory and proof assistants. I am also thankful to the Max Planck Institute for Informatics and all its member's staff for its economic and academic support during my Master's study.

Ich möchte den Freunden danken, die ich während der Zeit in Deutschland kennengelernt habe; Alexis, Anton, Gilbert, Hema, Ibung, Jannis, Jianlin, Lucia, Martin, Munem, Ricardo, Taqi, Viet, Zack. Vielen Dank an euch nicht nur für viele Spiel- und Filmabende, indem wir mit Bier so gesoffen waren, sondern auch für Kaffeepausen und Mittagessen in Mensa. Die waren richtig lustige Ablenkungen gegen Stress für mich. Ich werde diese Zeit niemals vergessen.

Finalmente pero no menos importante, quiero agradecer a mi familia, que a pesar de la distancia me han brindado su apoyo incondicional. A mis padres, Lorenza y Roberto, si soy quién soy es gracias a ustedes. A mis hermanas, Alma y Araceli, por el buen ánimo y apoyo. A las niñas, Tonche y Mina, por su ternura y alegría. De ustedes he aprendido mucho, gracias por creer en mi.

Contents

Chapter 1

Introduction

In this thesis we formalize and mechanize the proof of the undecidability of subtyping in System F by Pierce [33]. The proof is formalized in the Calculus of Inductive Constructions (CIC) [12, 31], which underlies the Coq proof assistant [42] where the mechanization is done. The mechanization constitutes a contribution to the Coq Library of Undecidability Proofs [22]. The browsable Coq code can be found in `www.ps.uni-saarland.de/~alvarez/coq/toc.html`. The definitions and theorems given in this work link to their mechanized counterpart. The Coq source code can be found in github.com/uds-psl/coq-undecidability-subtyping.

The notion of bounded quantification was first introduced by Cardelli and Wegner [9] in the system now called *kernel* System $F_{\leqslant:}$. A more general variant of the system was introduced by Curien and Ghelli [13], where they gave a type checking algorithm by means of a normalizing rewriting system of derivations. The algorithm is sound and complete by construction although it may loop. In his PhD thesis Ghelli [24] gave a proof of termination of the type checking algorithm, the proof was wrong and later Ghelli himself gave a counter example [25]. Based on Ghelli's non-terminating example Pierce [33] encoded two-counter machines in subtyping statements which shows that subtyping is indeed undecidable. The erroneous claim of termination illustrates how the subtleties in the proofs of computability properties may be overlooked, which is a strong argument in favor of machine-checked undecidability proofs and their ongoing development, for instance in the Coq Library of Undecidability Proofs (from now on abbreviated as the Undecidability Library).

Bounded quantification combines two orthogonal features ubiquitous among most programming languages, namely parametric polymorphism which forms the theoretical basis of generic and functional programming, and subtyping which is particularly important in object-oriented programming where a *behavioral* notion of subtyping [29] underlies the notion of inheritance.

Parametric polymorphism corresponds to the *terms depending on types* axis of the lambda cube [2], which organizes the three kinds of dependencies between types and terms that can extend the simply typed λ-calculus, meaning that polymorphism allows type abstractions at the term level. The other two axes are *types depending on types* corresponding to type operators [32, Ch. 29], and *types depending on terms* corresponding to dependent types [32, Sect. 30.5].

Parametric polymorphism is the core feature of System F, discovered independently by Girard [26] and Reynolds [35]. In System F there are type variables with type binders and type applications in addition to term variables with their binders and applications. This permits to write terms and types uniformly over arbitrary types. For example the *polymorphic* identity function can be written as $\Lambda\alpha.\lambda x : \alpha.\ x$ where Λ is a type binder and α a type variable, the function can be given the *polymorphic* type $\forall\alpha.\alpha \to \alpha$, where $\forall$ is a quantifier over types.

Meanwhile, subtyping is a relation of types written $\sigma{\leqslant}{:}\tau$, informally it means that a term of type σ can be used wherever a term of type τ is expected, σ is called a **subtype** and τ a **supertype**. Therefore well-typed terms have multiple types, namely all the supertypes of its given type, this is established by the rule called **subsumption**. A formal treatment of subtyping was given first by Reynolds [36], and later by Cardelli [8].

As a simple example consider the naturals and integers, in many programming languages $\mathbb{N}{\leqslant}{:}\mathbb{Z}$ reflecting the mathematical intuition that every natural number is also an integer. More interestingly, consider the following record types $\{\alpha : \tau, \beta : \sigma\}{\leqslant}{:}\{\alpha : \tau\}$; the record type with two fields is a subtype of the record type with a single field, and not the other way around. To see why consider how one may use the terms of record types; we can always *forget* a field of the record, but we cannot give a value to an arbitrary field. Therefore the supertypes of records have a subset of the original fields and the subtypes have a superset.

As a further example consider arrow types, to check that $\mathbb{Z} \to \mathbb{N}{\leqslant}{:}\mathbb{N} \to \mathbb{Z}$ we have to show how a given function of the type on the left can be used as a function of the type on the right. To apply the function of type $\mathbb{Z} \to \mathbb{N}$ to the argument of type $\mathbb{N}$ we need to be able to use it in place of the argument of type $\mathbb{Z}$, that is $\mathbb{N}{\leqslant}{:}\mathbb{Z}$. Additionally, once we have the result of type $\mathbb{N}$ we need to use it in place of the final result of type $\mathbb{Z}$, which again follows from $\mathbb{N}{\leqslant}{:}\mathbb{Z}$. In summary to check that $\mathbb{Z} \to \mathbb{N}{\leqslant}{:}\mathbb{N} \to \mathbb{Z}$ it is enough to check that $\mathbb{N}{\leqslant}{:}\mathbb{Z}$. Note how the first arguments of the arrow types are compared the other way around, therefore subtyping arrow types is said to be **contravariant** on their first argument.

Bounded quantification allows type abstraction over subtypes of a given bound, e.g. $\forall_{\alpha \leqslant : \sigma}.\tau$ denotes the family of polymorphic types τ that may be instantiated with subtypes of σ. This implies that in order for type instantiation to type check there is a subtype premise that needs to be fulfilled. For example, terms of type $\forall_{\alpha \leqslant : \{\beta : \sigma\}}.\tau$ may be instantiated with record types containing *at least* a field σ. Although illustrating, record types are unnecessary to show the undecidability of the subtype problem. In fact subtyping bounded quantifiers turns out to be complicated enough due to the *rebounding* of types explained in the next section. Therefore record subtyping is not discussed further.

On paper Pierce's proof is presented with named variables such as x and x_i, and assumptions of freshness are left implicit. Substitutions, like $[a/x]$ are also assumed to be capture-avoiding. Formalizing the proof in the named presentation is tedious, as such assumptions have to be explicitly verified. Mechanizing such proof is even harder, as for example, terms are considered identical up to α-equivalence.

The mechanization given in the present work is built around the syntax provided by the Autosubst 2 tool [40], as it provides a number of very useful Coq tactics and lemmas simplifying significantly the development. As a consequence the formalization is made with de Brujin indices and parallel substitutions [14].

Outline

The rest of this chapter elaborates on the subtyping problem and gives an introduction to *synthetic undecidability* which is the approach of the formalized proof, additionally we give an overview of related work and the relevant state of the art. In Chapter 2 we give some basic definitions of functions and data types used throughout the rest of the thesis. In Chapter 3 we give examples of *unscoped*, *well-scoped* and *polyadic* variants of syntax and show how to encode them appropriately, this illustrates one of the main sources of difficulties that arise in the mechanization due to the choice of deBrujin indices. In Chapter 4 we introduce the machine models relevant to the proof; two-counter machines and rowing machines, and show that the halting problem of rowing machines is undecidable. In Chapter 5 we give the formalization of the proof by Pierce [33] of the undecidability of the subtyping problem by introducing a series of refinements to the subtyping system, additionally we show that type checking is also undecidable which Pierce only mentions without much detail. In Chapter 6 we review some techniques used in the mechanization that, although unrelated to the mathematical argument, might be of interest to further developments and showcase some limitations in the inductive and recursive principles generated automatically by the Coq proof assistant, as well as a technique to consider syntax without specific constructs, specifically the syntax of types without arrows.

1.1 Subtyping Problem

Bounded quantification incorporates subtyping into the universal quantifier of System F. The type $\forall_{\alpha\leqslant:\sigma}.\tau$ contains terms of type τ abstracted over a type variable α, however unlike the regular type abstraction this terms can only be instantiated with subtypes of the provided upper bound σ. It is easy to see how we can recover unbounded quantification by introducing a new type $\top$ that is a trivial supertype of all types and using it as the upper bound.

$$\frac{\Gamma \vdash \tau_1 \leqslant: \sigma_1 \qquad \Gamma, \alpha \leqslant: \tau_1 \vdash \sigma_2 \leqslant: \tau_2}{\Gamma \vdash \forall_{\alpha\leqslant:\sigma_1}.\sigma_2 \leqslant: \forall_{\alpha\leqslant:\tau_1}.\tau_2}$$

Figure 1.1: Subtyping rule of the bounded quantifier

In a similar manner to arrow types, subtyping of bounded quantifiers is contravariant in the bounds; in order to check whether $\forall_{\alpha\leqslant:\sigma_1}.\sigma_2 \leqslant: \forall_{\alpha\leqslant:\tau_1}.\tau_2$ we have to be able to instantiate terms of the type on the left with types that can be used to instantiate terms with the type on the right, that is subtypes of τ_1, therefore $\tau_1 \leqslant: \sigma_1$.

However, unlike with arrow types, when checking the subtyping of the bodies in the quantifier we have to store the type variable in the context and this type variable has to have the tighter bound τ_1, as it corresponds to a type that can be used to instantiate both sides (Fig. 1.1).

As the type σ_2 gets a stricter constraint for its type variable α, it is said to be **rebounded** with τ_1. This allows transmission of *data* previously stored in the type on the right to be accessed by the type on the left, and in fact this faculty is what makes subtyping undecidable as it can be shown that the system without it (e.g. with the rule in Fig. 1.2) has a decidable subtype relation.

$$\frac{\Gamma, \alpha \leqslant: \phi \vdash \sigma \leqslant: \tau}{\Gamma \vdash \forall_{\alpha\leqslant:\phi}.\sigma \leqslant: \forall_{\alpha\leqslant:\phi}.\tau}$$

Figure 1.2: Restricted subtyping rule of the bounded quantifier

The restricted version of the rule forces the bound in the quantifiers to be identical, so the body of the type on the left is no longer rebounded and no *data* gets transmitted. The system with this restricted rule is called kernel System $F_{\leqslant:}$ [9][32, Sect. 28.3]. This system was studied first however, being less expressive, it is not discussed further in this work.

1.2 Synthetic Undecidability

Mathematically, in a **synthetic** approach the objects of interest are taken as primitives and are studied by formulating suitable axioms that capture their properties. This contrasts with the *analytic* approach where the objects are constructed from foundational primitives and their properties deduced.

Synthetic Computability theory was pioneered by Richman and Bridges [37, 7] and later developed by Bauer [3, 4]. The relevant axiom regarding computable functions is CT (from Church's thesis [37]) stating that every total function is computable. Saliently, every definable function in CIC is computable and total. For a detailed study on Synthetic Computability in CIC read Forster's discussion [22, Pt. 1].

Having only computable functions in our setting informs the definition of synthetic undecidability in several ways. Defining synthetic undecidability as the negation of decidability does not work as any predicate may be consistently assumed to be decidable. One may replace the falsity in that definition by the decidability or co-enumerability of the halting problem Halt, as they are unprovable and such definitions would transport undecidability along many-one reductions. Furthermore, replacing using co-enumerability instead of decidability yields a strictly weaker definition as decidability implies it while the converse requires classical reasoning. Hence *synthetic undecidability relative to a problem* Halt is defined for a problem P as:

$$\mathsf{decidable}\ P \to \mathsf{enumerable}\ \overline{\mathsf{Halt}}$$

With this definition, the halting problem of Turing machines can be shown synthetically undecidable relative to itself. From now on when we say that a problem is undecidable we mean that it is synthetically undecidable relative to the halting problem of Turing machines, as there are no other notions of undecidability used in the present work.

Most undecidability results in the literature are shown by reduction instead; a problem already shown undecidable is reduced to the problem in question. This technique lends itself nicely to the synthetic approach [21]. Concretely, many-one reductions transport decidability backwards. Therefore to show that a problem P is undecidable, given that Q is already shown undecidable, it is sufficient to show:

$$Q \preceq_m P$$

Pierce reduces the halting problem of two-counter machines (2CM halting) as defined in Chapter 4.1 to the subtyping problem ($F_{\leqslant:}$ subtyping). Dudenhefner [16] presents a variant of 2CM halting that is already mechanized in the Undecidability Library. Therefore to show that the subtyping problem is undecidable we just have to show:

$$\mathsf{2CM\ halting} \preceq_m F_{\leqslant:}\ \mathsf{subtyping}$$

1.3 Related Work

The paper by Pierce [33] constitutes the main source of the present thesis and contains the complete mathematical argument. However there are further published works that are important to the development and understanding of the proof. Pierce's own book *Types and programming languages* [32] provides a very thorough analysis of subtyping in System F in addition to the subtyping of existential, record, and object types. Pierce's proof uses the undecidability of the halting problem of two-counter machines shown by Minsky in 1967 [30]. The undecidability of the halting problem of a slightly different variant of two-counter machines is mechanized by Dudenhefner [16] by a reduction from the halting problem of Turing machines via Post's correspondence problem [34, 20]. as part of the Undecidability Library [22].

With regards to related mechanized undecidability results present in the Undecidability Library, Dudenhefner [17] also shows the undecidability of type checking in System F, in addition to typability and type inhabitation. The undecidability of higher-order unification on the *simply typed* λ-calculus [38], and the undecidability of the halting problem of the *untyped* λ-calculus [18] are also already mechanized. Additionally, the technique of modularized syntax used here (Sect. 6.3) is based on the one used in the mechanized proof of the undecidability of validity, satisfiability, and provability in First Order Logic [21]. For a summary of undecidability results present in the Library read Forster's overview [22, Ch. 25].

The mechanization in the present work is based in part on the code of Chapter 10 of Stark's work [39]. The code presented there is also a mechanization of System $F_{\leqslant:}$ using Autosubst 2 [40] to handle the underlying theory of variables and substitutions. The Autosubst 2 tool is used prominently in this work as there are several instances of syntax implemented with de Brujin indices and parallel substitutions [14], and the tool provides some very useful equational laws and simplification tactics.

The POPLmark (Principles of Programming Languages benchmark) challenge [1] is a set of problems intended to demonstrate the effectiveness of proof assistants in programming language theory. It concerns the so-called *algorithmic* subtyping relation in System F, which Pierce calls **Normal** System $F_{\leqslant:}$ (See 5.2). The first part of the challenge consist of showing that the system admits transitivity (which we show here in Lemma 5.4) and showing that the system augmented with record types is still transitive. Part 2 concerns type safety with and without pattern matching. Finally, Part 3 asks to develop the operational semantics of the previous systems. There are many submitted solutions to the challenge using a variety of proof assistants and mechanization techniques, we mention here some of them classifying them by how the syntax is formalized.

Presentations of the system using deBrujin indices, as it is here, are given by Stark [39] as well as Stump [41] and Vouillon [43] using the Coq proof assistant, Berghofer [5] using the Isabelle/HOL proof assistant, and Dicolla [15] using the VeriCode Proof Tool. Locally nameless presentations are given by Charguéraud [10] and Leroy [28] using the Coq proof assistant. Lastly, Ciaffaglione [11] gives a presentation using Higher-Order Abstract Syntax, also with the Coq proof assistant.

Finally, there are some results that are direct consequences of Pierce's result. Wehr and Thiemann [44] show the undecidability of subtyping bounded existential types from the the undecidability of subtyping in the *deterministic* system (See 5.7). The bounded existential types serve as a formalization of interface types as well as wildcards in the language JavaGI [45], where the existentially quantified variable has either a lower or upper type bound. Hu and Lhotak [27] show the undecidability of type checking and subtyping in the Dependant Object Types calculus, which formalizes the Scala type system. They use a similar technique to Pierce defining a *normal* variant of the type system which is the target of a reduction from the normal system defined by Pierce (See 5.2).

Chapter 2

Preliminaries

A number of notation for certain types is assumed throughout this thesis. The Calculus of Inductive Constructions has a hierarchy of type universes, however as they are not relevant to the present work, we consider a single type universe $\mathbb{T}$. Meanwhile $\mathbb{P}$ is the inpredicative universe of propositions. The trivial propositions are $\mathsf{True}, \mathsf{False} : \mathbb{P}$.

Analogous to propositions, the boolean type has two constructors $\mathsf{true}, \mathsf{false} : \mathbb{B}$. For the natural numbers $\mathbb{N}$ we use the standard notation for all numbers $1, 2$, etc., functions like addition $+$, multiplication $\times$, modulo $\%$, integer division $/$, and relations like $<$ and $\leq$.

We have a basic container data type which either stores a single value of an arbitrary type or *optionally* no value at all. The type argument is omitted in the constructors as it can be always inferred.

Definition 2.1 (Optional) $\mathbb{O} : \mathbb{T} \to \mathbb{T}$ *with constructors:*

$$\mathsf{None} : \forall X : \mathbb{T}.\ \mathbb{O}\ X$$
$$\mathsf{Some} : \forall X : \mathbb{T}.\ X \to \mathbb{O}\ X$$

The following data type has a *finite* variable number of values. It is one of the most important data types used as it serves as an implementation of scoped variables (see 3.1).

Definition 2.2 (Finite) $\mathbb{F} : \mathbb{N} \to \mathbb{T}$ *with superscript notation* $\mathbb{F}^n := \mathbb{F}\ n$ *has constructors:*

$$\mathsf{F}_0 : \forall n : \mathbb{N}.\ \mathbb{F}^{\mathsf{S}n}$$
$$\mathsf{F}_\mathsf{S} : \forall n : \mathbb{N}.\ \mathbb{F}^n \to \mathbb{F}^{\mathsf{S}n}$$

The natural number argument of the constructors is omitted. To reason about values of these we would like a bijection with the *bounded* naturals. The injection from finite types into the naturals is obvious, it maps the constructors appropriately.

Definition 2.3 $\mathsf{fin2nat} : \forall n : \mathbb{N}.\ \mathbb{F}^n \to \mathbb{N}$ *with notation* $\widehat{x} := \mathsf{fin2nat}\ \_\ x$ *is defined as:*

$$\widehat{\mathsf{F}_0} := 0$$
$$\widehat{\mathsf{F_S}\ x} := \mathsf{S}\ \widehat{x}$$

moreover if $x : \mathbb{F}^n$ *then* $\widehat{x} < n$.

The inverse of the previous function has to be guarded, it sends natural numbers to a finite type a larger size.

Definition 2.4 $\mathsf{nat2fin} : \forall n, m : \mathbb{N}.\ n < m \to \mathbb{F}^m$ *with notation* $n^\circ := \mathsf{nat2fin}\ n\ \_\ \_$ *is defined as:*

$$0^\circ := \mathsf{F}_0$$
$$(\mathsf{S}n)^\circ := \mathsf{F_S}\ n^\circ$$

We use the $-^\circ$ *notation when the inequality hypothesis can be determined from context. Moreover for all* $m, n : \mathbb{N}$ *if* $n < m$ *then* $\widehat{n^\circ} = n$.

Finally, we also require a container data type of arbitrary size. As the size of the container is part of the argument we opt to use vectors instead of another kind of array.

Definition 2.5 (Vectors) $\mathbb{V} : \mathbb{T} \to \mathbb{N} \to \mathbb{T}$ *with superscript notation* $X^n := \mathbb{V}\ X\ n$ *has constructors:*

$$\mathsf{Nil} : \forall X : \mathbb{T}.X^0$$
$$\mathsf{Cons} : \forall X : \mathbb{T}, n : \mathbb{N}.\ X \to X^n \to X^{\mathsf{S}n}$$

If ambiguity with other superscript notation arises extra parenthesis may be used. For constructors we omit the type argument X and the length argument, we use the standard notation $[\,] := \mathsf{Nil}\ \_\ \_$ and $x :: V := \mathsf{Cons}\ \_\ \_ x\ V$. Vectors may be accessed safely with an element of a finite type of the same size as the vector.

Definition 2.6 $\mathsf{nth'} : \forall X : \mathbb{T},\ n : \mathbb{N}.\ X^n \to \mathbb{F}^n \to X$ *with notation* $V[n] := \mathsf{nth'}\ \_\ \_\ V\ n$ *is defined as:*

$$(x :: V)\ [\mathsf{F}_0] := x$$
$$(x :: V)\ [\mathsf{F_S}\, f] := V[f]$$

Note that to access $[\,]$ we need an argument of type $\mathbb{F}^0$ which is empty, therefore we cannot get an element from the empty vector, as expected.

2.1 Synthetic Undecidability

In this section we give several basic definitions of synthetic computability [21]. The starting point is a definition of decidability. Exploiting the fact that every function definable in CIC is computable we may define decidability of a predicate as the existence of a **reflecting** function.

Definition 2.7 (Decidability) *For all $X : \mathbb{T}$ and predicate $P : X \to \mathbb{P}$:*

$$\mathsf{decidable}\ P := \exists f : X \to \mathbb{B}.\ \forall x : X.\ P\ x \leftrightarrow f\ x = \mathsf{true}$$

A type X is **discrete** if it has decidable equality, i.e. $\mathsf{decidable}\ (\lambda x, y : X.\ x = y)$. A predicate is called enumerable if the set of values that fulfills it is either empty or the range of a total function from the naturals, this case distinction is established using the optional type as all CIC functions are total.

Definition 2.8 (Enumerability) *For all $X : \mathbb{T}$ and predicate $P : X \to \mathbb{P}$:*

$$\mathsf{enumerable}\ P := \exists f : \mathbb{N} \to \mathbb{O}X.\ \forall x : X.\ P\ x \leftrightarrow \exists n.\ f\ n = \mathsf{Some}\ x$$

A type X is called **enumerable** if it has an enumerator, i.e. $\mathsf{enumerable}\ (\lambda x : X.\ \mathsf{True})$. From the previous definitions we arrive at the *synthetic* definition of undecidability relative to the halting problem Halt.

Definition 2.9 (Synthetic Undecidability) *For all $X : \mathbb{T}$ and predicate $P : X \to \mathbb{P}$:*

$$\mathsf{undecidable}\ P := \mathsf{decidable}\ P \to \mathsf{enumerable}\ \overline{\mathsf{Halt}}$$

We will show undecidability by constructing a *many-one reduction* which converts instances of a problem to instances of another problem in an appropriate manner.

Definition 2.10 (Many-one reduction) *For all $X, Y : \mathbb{T}$, and predicates $P : X \to \mathbb{P}$, and $Q : Y \to \mathbb{P}$, P is many-one reducible to Q if:*

$$P \preceq_m Q := \exists f : X \to Y.\ \forall x : X.\ P(x) \leftrightarrow Q(f(x))$$

It is easy to check that reductions transport decidability and enumerability backwards. From this fact it follows that, given the undecidability of a problem, we can conclude *undecidability from reducibility*.

Fact 2.11 *For all $X, Y : \mathbb{T}$, and predicates $P : X \to \mathbb{P}$, $Q : Y \to \mathbb{P}$, such that $P \preceq_m Q$:*

- $\mathsf{decidable}\ Q \to \mathsf{decidable}\ P$
- *if X is enumerable and Y is discrete then* $\mathsf{enumerable}\ Q \to \mathsf{enumerable}\ P$

Fact 2.12 (Undecidability from reducibility) *Given predicates such that $P \preceq_m Q$:*

$$\mathsf{undecidable}\ P \to \mathsf{undecidable}\ Q$$

Chapter 3

Aspects of de Brujin Syntax

The formalizations of the syntax that use substitution are defined using de Brujin **indices** [14], where a variable is identified with the number of binders between that occurrence and its corresponding binder. For example $\lambda x.(\lambda y.\ yx)\ x$ is represented by $\lambda.(\lambda.0\ 1)\ 0$. Substitutions are therefore functions from the naturals to terms, and instantiation of variables is just function application.

The Autosubst 2 tool [40] produces the mechanization of de Brujin syntax; it generates an inductive type of terms, a number of primitive substitutions, and some lemmas stating the laws of instantiation. Autosubst 2 is used extensively throughout this thesis as it provides features necessary for some of the syntax used.

Autosubst 2 offers **well-scoped** [6] presentations of de Brujin syntax where terms carry an upper bound to the free variables, called a scope. Consequently, a term used in the wrong scope does not type check. Additionally, scoped syntax greatly simplifies stating certain lemmas while maintaining technical details. In contrast we call the presentation without bounds **unscoped**.

An additional feature of Autosubst 2 is **polyadic** binders [40], which bind an arbitrary number of variables simultaneously. Note that a polyadic binder is different from a variadic binder as defined by Stark [39], which is an abbreviation of n unary binders; in a polyadic binder the variables are introduced simultaneously with the same scope. A well-scoped presentation shows explicitly how the variables have the same scope and therefore cannot refer to each other.

In this chapter we show how translations of syntaxes using different features of Autosubst 2 can be defined in a coherent manner. The mechanization in Chapter 5 involves a translation that requires careful thought; in particular when translating a binder that is annotated with a term that has the same free variables.

Examples of syntax consisting only of binders and variables are presented to avoid clutter, however the problem becomes trivial in some syntaxes. In particular if the

binders are not annotated, or the term annotation does not include variables of the same sort as the binder itself, then any variadic binder is trivially polyadic.

In general the binders of second order theories are not trivially polyadic, as long as the term annotation is of the same syntactic sort as the binder. Consider the $\forall$ binder of System $F_{\leqslant:}$; its term annotation is the upper bound of the type variable which is also a type. Note that the Π and Σ binders of dependent type theories are also not trivially polyadic by the same reason.

In the following sections we start by defining the terms and substitutions of the syntax, in addition to renamings which are substitutions that replace variables with variables. Next we define the encodings of types and substitutions an show that instantiation commutes with encoding.

3.1 Well-scoped and Unscoped Syntaxes

The syntax only includes type variables and type quantifiers as those are the relevant syntactic constructs, therefore it is a subset of the syntax of System $F_{\leqslant:}$.

Definition 3.1 *The syntax of unscoped types is defined by the following grammar:*

$$\tau, \sigma : \mathsf{Type} ::= \ x \mid \forall \sigma.\ \tau \qquad x : \mathbb{N}$$

The scope of a term of this syntax can be recovered by the following predicate:

Definition 3.2 *The predicate* $\mathsf{closed} : \mathsf{Type} \to \mathbb{N} \to \mathbb{P}$ *is defined inductively:*

$$\frac{x < n}{\mathsf{closed}\ (x)\ n} \qquad \frac{\mathsf{closed}\ \sigma\ n \qquad \mathsf{closed}\ \tau\ (\mathsf{S}n)}{\mathsf{closed}\ (\forall \sigma.\tau)\ n}$$

Renamings are instances of substitutions, when instantiated a renaming replaces variables with variables.

Definition 3.3 (Renamings) *An unscoped renaming is a function* $\xi : \mathbb{N} \to \mathbb{N}$*, the primitive renamings are defined as follows:*

- ***Shift:*** $\uparrow := \mathsf{S}$
- ***Extension:*** $(y \cdot \xi)(x) := \begin{cases} y & \text{if } x = 0 \\ \xi(n) & \text{if } x = \mathsf{S}\ n \end{cases}$
- ***Lift:*** $\Uparrow \xi := 0 \cdot (\xi \circ \langle\uparrow\rangle)$

Note that the lifting of a renaming is 0 exactly when the argument is 0, this has the effect of leaving the variable 0 unchanged. Meanwhile the renaming is applied to the preceding variable in the non-zero case, as it should ignore the new variable over which the renaming is being lifted, and is shifted to the new scope.

Definition 3.4 (Instantiation of renamings) *For all* $t : \mathsf{Type}$ *and* $\xi : \mathbb{N} \to \mathbb{N}$ *the instantiation is defined recursively:*

$$(\forall\sigma.\tau)\langle\xi\rangle := \forall\sigma\langle\xi\rangle.\tau\langle\Uparrow\xi\rangle$$
$$(x)\langle\xi\rangle := \xi(x)$$

When instantiated a substitution replaces variables with terms. A substitution can be obtained from a renaming by composition with the variable constructor, therefore we consider it a primitive substitution.

Definition 3.5 (Substitutions) *An unscoped substitution is a function* $\theta : \mathbb{N} \to \mathsf{Type}$*, the primitive substitutions are defined as follows:*

- ***Var****: the constructor for variables* $\mathbb{N} \to \mathsf{Type}$.
- ***Extension****:* $(t \cdot \theta)(x) := \begin{cases} t & \text{if } x = 0 \\ \theta(n) & \text{if } x = \mathsf{S}\ n \end{cases}$
- ***Lift****:* $\Uparrow\theta\ (x) := 0 \cdot (\theta \circ [\uparrow])$

Again the lifting of a substitution has the effect of leaving the type variable 0 unchanged and applying the substitution shifted once.

Definition 3.6 (Instantiation of substitutions) *For all* $t : \mathsf{Type}$ *and* $\theta : \mathbb{N} \to \mathsf{Type}$ *the instantiation is defined recursively:*

$$(\forall\sigma.\tau)[\theta] := \forall\sigma[\theta].\tau[\Uparrow\theta]$$
$$(x)[\theta] := \theta(x)$$

In addition to the syntax types Autosubst 2 provides a number of equations involving the substitution primitives. We present here the subset of those equations called the *Interaction Laws* which are the most general, although several other more specific results are also given by the tool.

Lemma 3.7 *Interaction Laws*

- *Identity:* $\mathsf{id} \circ f = f = f \circ \mathsf{id}$
- *Associativity:* $(f \circ g) \circ h = f \circ (g \circ h)$
- *Interaction:* $\uparrow \circ (t \cdot \theta) = \theta$
- *Distributivity:* $(t \cdot \theta) \circ f = f(t) \cdot (\theta \circ f)$
- *η-identity:* $0 \cdot \uparrow = \mathsf{id}$
- *η-law:* $\theta(0) \cdot (\uparrow \circ \theta) = \theta$

Note that some of them are trivial corollaries of the definition of the primitives. Instances of these laws are given for well-scoped syntax however they are not discussed further.

We turn our attention to the well-scoped presentation of the same syntax. The primitives of renamings and substitutions change definition, instantiation is identical with the new definitions so it is omitted.

Definition 3.8 *The syntax of well-scoped types is defined by the following grammar:*

$$\tau^n, \sigma^n : \mathsf{Type}^n ::= \; x \mid \forall \sigma^n.\, \tau^{\mathsf{S}n} \qquad x : \mathbb{F}^n$$

Definition 3.9 (Renamings) *For all $m, n : \mathbb{N}$ a well-scoped renaming is a function $\xi : \mathbb{F}^n \to \mathbb{F}^m$, the primitive renamings are defined as follows:*

- ***Shift**:* $\uparrow := \mathsf{F_S}$
- ***Lift**:* $\Uparrow \xi\ (x) := \begin{cases} \mathsf{F}_0 & \text{if } x = \mathsf{F}_0 \\ \mathsf{F_S}(\xi\ n) & \text{if } x = \mathsf{F_S}\ n \end{cases}$

Definition 3.10 (Substitutions) *For all $m, n : \mathbb{N}$ a well-scoped substitution is a function $\theta : \mathbb{F}^n \to \mathsf{Type}^m$, the primitive substitutions are defined as follows:*

- ***Var**: the constructor for variables $\mathbb{F}^n \to \mathsf{Type}^n$.*
- ***Lift**:* $\Uparrow \xi\ (x) := \begin{cases} \mathsf{F}_0 & \text{if } x = \mathsf{F}_0 \\ (\theta\ n)\langle\uparrow\rangle & \text{if } x = \mathsf{F_S}\ n \end{cases}$

3.1.1 Encoding

We define the encoding between the terms of the syntaxes. The encoding of variables is just the injection into the naturals.

Definition 3.11 *For all $n : \mathbb{N}$ the encoding $[\![-]\!] : \mathsf{Type}^n \to \mathsf{Type}$ is defined as:*

$$\begin{aligned} [\![\forall \sigma^n.\tau^{\mathsf{S}n}]\!] &:= \forall [\![\sigma^n]\!].[\![\tau^{\mathsf{S}n}]\!] \\ [\![x]\!] &:= \widehat{x} \end{aligned}$$

Definition 3.12 *For all $n, m : \mathbb{N}$, and any well-scoped renaming $\xi : \mathbb{F}^n \to \mathbb{F}^m$ the encoding is an unscoped renaming $[\![\xi]\!] : \mathbb{N} \to \mathbb{N}$ defined as:*

$$[\![\xi]\!](x) := \begin{cases} \widehat{\xi(x^\circ)} & \text{if } x < n \\ x & \text{otherwise} \end{cases}$$

In order to show that the encoding of renamings commutes with instantiation we need the following *bounded extensionality* result.

Fact 3.13 *For all $n : \mathbb{N}$, $t : \mathsf{Type}$ and renamings $\xi, \xi' : \mathbb{N} \to \mathbb{N}$ if* closed *$t\ n$ and $\xi(x) = \xi'(x)$ for all $x < n$, then $t\langle\xi\rangle = t\langle\xi'\rangle$.*

Corollary 3.14 (Extensionality up to n) *For any $n : \mathbb{N}$, and unscoped renamings ξ, ξ' if $\xi(x) = \xi'(x)$ for all $x < n$ then for any type $t : \mathsf{Type}^n$:*

$$[\![t]\!]\langle\xi\rangle = [\![t]\!]\langle\xi'\rangle$$

Proof. By fact 3.13 we just have to check that $\mathsf{closed}\ [\![t]\!]\ n$ which is trivial.

Lemma 3.15 *For all n, m, any well-scoped type $t : \mathsf{Type}^n$, and well-scoped renaming $\xi : \mathbb{F}^n \to \mathbb{F}^m$:*

$$[\![t\langle\xi\rangle]\!] = [\![t]\!]\langle[\![\xi]\!]\rangle$$

Proof. By induction on t with extensionality up to n. □

We can encode substitutions using the same technique.

Definition 3.16 *For all m, n, and well-scoped substitution $\theta : \mathbb{F}^n \to \mathsf{Type}^m$ the encoding is an unscoped substitution defined as follows:*

$$[\![\theta]\!](x) := \begin{cases} [\![\theta(x^\circ)]\!] & \text{if } x < n \\ x & \text{otherwise} \end{cases}$$

Fact 3.17 *For all $n : \mathbb{N}$, $t : \mathsf{Type}$, and unscoped substitutions $\theta, \theta' : \mathbb{N} \to \mathsf{Type}$ if $\mathsf{closed}\ t\ n$ and $\theta(x) = \theta'(x)$ for all $x < n$ then $t[\theta] = t[\theta]$*

Corollary 3.18 (Extensionality up to n) *For any n, and unscoped substitutions θ, θ' if $\theta(x) = \theta'(x)$ for all $x < n$ then for any $t : \mathsf{Type}^n$:*

$$[\![t]\!][\theta] = [\![t]\!][\theta']$$

Lemma 3.19 *For any type $t : \mathsf{Type}^n$, and well-scoped substitution $\theta : \mathbb{F}^n \to \mathsf{Type}^m$:*

$$[\![t[\theta]]\!] = [\![t]\!]\ [[\![\theta]]\!]$$

Proof. By induction on t with extensionality up to n. □

3.2 Polyadic Binders

In this section we consider a minimal syntax with only a polyadic binder. The goal is to encode it as the well-scoped syntax of the previous section. We start by defining the polyadic syntax which is a subset of the polarized syntax (Def. 5.6) of Systems $F^D_{\leqslant:}$ and $F^F_{\leqslant:}$. We say that a polyadic binder is w-fold if it binds w fresh variables.

Definition 3.20 *For all $w : \mathbb{N}$ the syntax of well-scoped types with w-fold polyadic binders is defined by the following grammar:*

$$\tau^n : \mathsf{pType}^n_w ::= \ (x, i) \mid \forall \tau^n_1, \ldots, \tau^n_w.\ \tau^{\mathsf{S}n} \qquad x : \mathbb{F}^n, i : \mathbb{F}^w$$

The variables are now a pair of finite indices, one identifies the binder and the other identifies the specific term within that binder. In a type variable of pType^n_w we refer to the index of type $\mathbb{F}^n$ as the **inter**-binder component of the variable, while we refer to the index of type $\mathbb{F}^w$ as the **intra**-binder component. As renamings correspond to changing the order of binders they ignore the intra-binder component of variables.

Definition 3.21 (Renamings) *For all $m, n, w : \mathbb{N}$ a w-fold polyadic renaming is a function $\xi : \mathbb{F}^n \to \mathbb{F}^m$, the primitive renamings are defined as follows:*

- ***Shift***: $\uparrow (x, i) := (\mathsf{F_S} x, i)$
- ***Lift***: $\Uparrow \xi\ (x, i) := \begin{cases} (\mathsf{F}_0, i) & \text{if } x = \mathsf{F}_0 \\ (\mathsf{F_S}(\xi\ y), i) & \text{if } x = \mathsf{F_S}\ y \end{cases}$

To instantiate a renaming we apply it to the inter-binder component of variables.

Definition 3.22 (Instantiation of renamings) *For all $m, n, w : \mathbb{N}$, $t : \mathsf{pType}^n_w$, and polyadic renaming $\xi : \mathbb{F}^n \to \mathbb{F}^m$, the instantiation $t\langle\xi\rangle : \mathsf{pType}^m_w$ is defined recursively:*

$$\begin{aligned} (\forall \tau_1, \ldots, \tau_w.\tau)\langle\xi\rangle &:= \forall \tau_1\langle\xi\rangle, \ldots, \tau_w\langle\xi\rangle.\tau\langle\Uparrow \xi\rangle \\ (x, i)\langle\xi\rangle &:= (\xi\ x, i) \end{aligned}$$

Polyadic substitution takes into account both components of the variables.

Definition 3.23 (Substitutions) *For all $m, n, w : \mathbb{N}$ a w-fold polyadic substitution is a function $\theta : \mathbb{F}^n \to \mathbb{F}^w \to \mathsf{pType}^m_w$, the primitive substitutions are defined as follows:*

- ***Var***: *the constructor for variables $\mathbb{F}^n \to \mathbb{F}^w \to \mathsf{pType}^n_w$.*
- ***Lift***: $\Uparrow \theta\ (x, i) := \begin{cases} (x, i) & \text{if } x = \mathsf{F}_0 \\ (\theta\ y\ i)\langle\uparrow\rangle & \text{if } x = \mathsf{F_S}\ y \end{cases}$

Definition 3.24 (Instantiation of substitutions) *For all $m, n, w : \mathbb{N}$, $t : \mathsf{pType}^n_w$ and $\theta : \mathbb{F}^n \to \mathbb{F}^w \to \mathsf{pType}^m_w$ the instantiation is defined recursively:*

$$\begin{aligned} (\forall \tau_1, \ldots, \tau_w.\tau)[\theta] &:= \forall \tau_1[\theta], \ldots, \tau_w[\theta].\ \tau[\Uparrow \theta] \\ (x, i)[\theta] &:= \theta\ x\ i \end{aligned}$$

3.2.1 Encoding

The encoding of the polyadic binders requires some consideration, after the encoding the first bound in the polyadic binder there must be a new bound variable so the encoding of the next bound is **shifted**. The shiftings accumulate so the nth bound is shifted $n - 1$ times.

Definition 3.25 *For all $n, w : \mathbb{N}$ the encoding $[\![-]\!] : \mathsf{pType}^n_w \to \mathsf{Type}^{n \times w}$ is defined as:*

$$[\![\forall \tau_1, \ldots, \tau_w.\tau]\!] := \forall [\![\tau_1]\!].\forall [\![\tau_2]\!]\langle\uparrow\rangle \ldots \forall [\![\tau_w]\!]\langle\uparrow^{w-1}\rangle.[\![\tau]\!]$$
$$[\![(x, i)]\!] := (w \times \widehat{x} + \widehat{i})^\circ$$

The encoding of variables combines both components as each inter-binder index corresponds to w intra-binder ones. Because renamings only modify the inter-binder component we need to split a combined variable, apply the renaming and then combine the result. We achieve this using integer division and modulo.

Definition 3.26 *For all $m, n, w : \mathbb{N}$, and any w-fold polyadic renaming $\xi : \mathbb{F}^n \to \mathbb{F}^m$ the encoding is a well-scoped renaming $[\![\xi]\!] : \mathbb{F}^{w \times n} \to \mathbb{F}^{w \times m}$ defined as:*

$$[\![\xi]\!](x) := (\widehat{\xi(\widehat{x}/w)^\circ} + \widehat{x}\%w)^\circ$$

In the case of substitutions we just need to split the combined variable so we can apply the substitution and encode the result.

Definition 3.27 *For all $m, n, w : \mathbb{N}$, and any w-fold polyadic substitution $\theta : \mathbb{F}^n \to \mathbb{F}^w \to \mathsf{pType}^m_w$ the encoding is a well-scoped substitution $[\![\theta]\!] : \mathbb{F}^{n \times w} \to \mathsf{Type}^{m \times w}$ defined as:*

$$[\![\theta]\!](x) := [\![\theta\ (\widehat{x}/w)^\circ\ (\widehat{x}\%w)^\circ]\!]$$

It is straightforward to show that this encoding commutes with instantiation, as extensionality is always "up to a bound".

Fact 3.28 *For all $m, n : \mathbb{N}$, $t : \mathsf{Type}^n$ and well-scoped renamings $\xi, \xi' : \mathbb{F}^n \to \mathbb{F}^m$, if $\xi(x) = \xi'(x)$ for all $x : \mathbb{F}^n$ then $t\langle\xi\rangle = t\langle\xi\rangle$*

Lemma 3.29 *For all $m, n, w : \mathbb{N}$, $t : \mathsf{pType}^n_w$ and w-fold polyadic renaming $\xi : \mathbb{F}^n \to \mathbb{F}^m$:*

$$[\![t\langle\xi\rangle]\!] = [\![t]\!]\langle[\![\xi]\!]\rangle$$

Proof. By induction on t. □

Fact 3.30 *For all $m, n : \mathbb{N}$, $t : \mathsf{Type}^n$ and well-scoped substitutions $\theta, \theta' : \mathbb{F}^n \to \mathsf{Type}^m$, if $\theta(x) = \theta'(x)$ for all $x : \mathbb{F}^n$ then $t[\theta] = t[\theta']$*

Lemma 3.31 *For all $m, n, w : \mathbb{N}$, $t : \mathsf{pType}^n_w$ and w-fold polyadic substitution $\theta : \mathbb{F}^{w \times n} \to \mathsf{pType}^m_w$:*

$$[\![t[\theta]]\!] = [\![t]\!][[\![\theta]]\!]$$

Proof. By induction on t. □

Chapter 4

Machine Models

In this chapter we describe the machine models involved in the proof of undecidability of subtyping. We introduce here *two-counter machines*, which is a well-known model with halting problem already shown undecidable. We also introduce *rowing machines*, which is a machine model intuitively closer to the subtyping systems defined in Chapter 5. The halting problem of two-counter machines is reduced to the halting problem of rowing machines, with which it is shown to also be undecidable.

4.1 Two-counter Machines

Pierce defines two-counter machines as consisting of a configuration and a program. The configuration is a triplet of natural numbers consisting of the program counter and two registers (PC, A, B), meanwhile a program is a sequence (of length w) of instructions.

Definition 4.1 *The type of two-counter machines is defined as:*

$$\mathsf{CM2}_w := \mathbb{F}^w \times \mathbb{N} \times \mathbb{N} \times (\mathsf{Inst}_w)^w$$

Where the type of instructions is defined by the following grammar:

$$\iota : \mathsf{Inst}_w ::= \mathtt{inc}_A n \mid \mathtt{inc}_B n \mid \mathtt{dec}_A n/m \mid \mathtt{dec}_B n/m \mid \mathtt{Halt} \qquad n, m : \mathbb{F}^w$$

We give an inductive definition of the step relation instead of the definition using a function $\mathsf{CM2}_w \to \mathbb{O}\mathsf{CM2}_w$ that is useful in the mechanized proof.

Definition 4.2 *Using infix notation the step relation* $\succ_C$: $\mathsf{CM2}_w \to \mathsf{CM2}_w \to \mathbb{P}$ *is defined inductively:*

$$\frac{P[n] = \mathtt{inc}_A n'}{(n,a,b,P) \succ_C (n', \mathsf{S}a, b, P)} \qquad \frac{P[n] = \mathtt{inc}_B n'}{(n,a,b,P) \succ_C (n', a, \mathsf{S}b, P)}$$

$$\frac{P[n] = \mathtt{dec}_A n'/m'}{(n,0,b,P) \succ_C (n', 0, b, P)} \qquad \frac{P[n] = \mathtt{dec}_B n'/m'}{(n,a,0,P) \succ_C (n', a, 0, P)}$$

$$\frac{P[n] = \mathtt{dec}_A n'/m'}{(n,\mathsf{S}a,b,P) \succ_C (m', a, b, P)} \qquad \frac{P[n] = \mathtt{dec}_B n'/m'}{(n,a,\mathsf{S}b,P) \succ_C (m', a, b, P)}$$

Definition 4.3 (CM2 halting) *A two-counter machine* M *halts if* $M \succ_C^* (n,a,b,P)$ *and* $P[n] = \mathtt{Halt}$.

Pierce's definition differs slightly from the one presented by Dudenhefner [16] in the Undecidability Library, however it is easy to check that they are equivalent.

In the library the machines are lists of instructions and the step function maps a machine and a configuration to a configuration. We use Pierce's machines so the reduction from Dudenhefner's machines serves as a preliminary in the final reduction to switch from lists and natural number indices to vectors and finite types.

Definition 4.4 *The type of Dudenhefner's two-counter machines is defined as:*

$$\mathsf{CM2}' := \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{L}(\mathsf{Inst}')$$

The instruction set is defined by the following grammar:

$$\iota : \mathsf{Inst}' ::= \mathtt{inc}_A \mid \mathtt{inc}_B \mid \mathtt{dec}_A n \mid \mathtt{dec}_B n \qquad n : \mathbb{N}$$

Definition 4.5 (CM2' halting) *A Dudenhefner's two-counter machine* M *halts if* $M \succ_{C'}^* (n,a,b,P)$ *and* $\mathsf{length}\ P < n$.

Theorem 4.6 CM2' halting *is undecidable.*

Instead of having a dedicated halting instruction, these machines halt when the PC is larger than the length of the program. Additionally, when the values in the registers are increased (or are 0 and the instruction is `dec`) the PC simply moves to the next instruction. The translation of a instruction depends on its position in the program in order to get the index of the next instruction.

Lemma 4.7 *A Dudenhefner's two-counter machine halts iff its translation as a Pierce's two-counter machine halts.*

Theorem 4.8 CM2 halting *is undecidable.*

4.2 Rowing Machines

Pierce [33] defines the rowing machines as consisting of a non zero number of registers, the first one used as program counter by convention. The only machine of zero registers is the empty vector, so we consider it as a separate trivial case.

Definition 4.9 *For any $w : \mathbb{N}$ such that $w > 0$ a **rowing machine of width** w is a vector, written $\langle \rho_1, \ldots, \rho_w \rangle : \mathsf{RM}_w$*

$$\mathsf{RM}_w := (\mathsf{Row}_w)^w$$

*Each ρ is a closed **row of width** w defined by the following grammar:*

$$\rho : \mathsf{Row}_w ::= \; x_i \mid \lambda_{x_1,\ldots,x_w}.\langle \rho_1, \ldots, \rho_w \rangle \mid \texttt{Halt} \qquad i \leq w$$

In the **abstract** row $\lambda_{x_1,\ldots,x_w}.\langle \rho_1, \ldots, \rho_w \rangle$ the variables x_i are bounded in all of the ρ_i. This w-fold binder is trivially polyadic as the variables are "untyped". When instantiated with w closed rows, an abstract row yields a rowing machine. The transition function of the rowing machine instantiates its program counter with the contents of the w registers. However instead of defining the step relation in terms of a function $\mathsf{RM}_w \to \mathbb{O}\ \mathsf{RM}_w$ like in the code, we give an inductive definition as the function is unnecessary in the high level argument.

Definition 4.10 *Using infix notation the step relation $\succ_R$: $\mathsf{RM}_w \to \mathsf{RM}_w \to \mathbb{P}$ is defined inductively:*

$$\frac{\rho_1 = \lambda_{x_1,\ldots,x_w}.\langle \rho'_1, \ldots, \rho'_w \rangle}{\langle \rho_1, \ldots, \rho_w \rangle \succ_R \langle \rho'_1[\rho_1/x_1, \ldots, \rho_w/x_w], \ldots, \rho'_1[\rho_1/x_1, \ldots, \rho_w/x_w] \rangle}$$

Definition 4.11 (RM halting) *A rowing machine M halts if for some $\rho_2, \ldots, \rho_w$*

$$M \succ_R^* \langle \texttt{Halt}, \rho_2 \ldots \rho_w \rangle$$

4.3 Undecidability

Pierce shows the undecidability of the halting problem of rowing machines by reduction from the halting problem of two-counter machines. In order to encode two-counter machines as rowing machines we need an encoding of instructions and of natural numbers in each register; $\mathcal{R}_A$ and $\mathcal{R}_B$. For a program of length w the encoding is a rowing machine of width $w + 5$, the first 5 registers store PC, A, B, and the branching addresses for the decreasing instruction, the last w registers store the encoding of the program.

The encoding of instructions $\mathcal{R} : \mathsf{Inst}_w \to \mathsf{Row}_{w+5}$ and the encodings of naturals $\mathcal{R}_A$,$\mathcal{R}_B$: $\mathbb{N} \to \mathsf{Row}_{w+5}$ are omitted here for brevity.

Definition 4.12 *The encoding of two-counter machines* $\mathcal{R} : \mathsf{CM2}_w \to \mathsf{RM}_{w+5}$ *is defined as:*

$$\mathcal{R}(n, a, b, (\iota_1, \ldots, \iota_w)) := \langle \mathcal{R}(\iota_n), \mathcal{R}_A(a), \mathcal{R}_B(b), \texttt{Halt}, \texttt{Halt}, \mathcal{R}(\iota_1), \ldots, \mathcal{R}(\iota_w) \rangle$$

Theorem 4.13 *M halts iff $\mathcal{R}(M)$ halts.*

Proof. ($\Rightarrow$) By induction on the given trace.
($\Leftarrow$) Assuming $\mathcal{R}(M) \succ_R^n \langle \texttt{Halt} \ldots \rangle$, by complete induction on n. We can assume that $M \succ_C M'$ for a two-counter machine M', otherwise the goal is trivial. By determinism of $\succ_C$ we can change the goal to M' halts, meanwhile by determinism of $\succ_R$ we have $\mathcal{R}(M) \succ_R^m \mathcal{R}(M') \succ_R^{n-m} \langle \texttt{Halt} \ldots \rangle$, for some $m > 0$. The proof is completed by the induction hypothesis with $\mathcal{R}(M') \succ_R^{n-m} \langle \texttt{Halt} \ldots \rangle$. □

Theorem 4.14 RM halting *is undecidable.*

Proof. By Lemma 2.12 and Theorem 4.13. □

Chapter 5

Subtyping and Type checking

In this chapter we give a reformulation of Pierce's result [33], namely the undecidability of the subtyping problem (Theorem 5.18), from which the undecidability of type checking follows (Theorem 5.21).

$$
\begin{aligned}
t, u : \mathsf{Term} &::= x \mid \lambda_{x:\tau}.\, t \mid t\ u \mid \Lambda_{x\leqslant:\tau}.\, t \mid t\ \tau \\
\tau, \sigma : \mathsf{Type} &::= \alpha \mid \sigma \to \tau \quad \mid \forall_{\alpha\leqslant:\sigma}.\, \tau \mid \top \\
\Gamma : \mathsf{Context} &::= [\,] \quad \mid \Gamma, \alpha\leqslant:\tau \\
\Delta : \mathsf{TermContext} &::= [\,] \mid \Delta, x : \tau
\end{aligned}
$$

$$\frac{x : \tau \in \Delta}{\Delta; \Gamma \vdash x : \tau}\ \text{Var} \qquad \frac{\Delta; \Gamma \vdash t : \sigma \qquad \Gamma \vdash \sigma\leqslant:\tau}{\Delta; \Gamma \vdash t : \tau}\ \text{Subsumption}$$

$$\frac{\Delta, x : \sigma; \Gamma \vdash t : \tau}{\Delta; \Gamma \vdash \lambda_{x:\sigma}.t\ : \sigma \to \tau}\ \text{Term-Abst} \qquad \frac{\Delta; \Gamma \vdash t : \sigma \to \tau \qquad \Delta; \Gamma \vdash u : \sigma}{\Delta; \Gamma \vdash t\ u : \tau}\ \text{Term-Inst}$$

$$\frac{\Delta; \Gamma, \alpha\leqslant:\sigma \vdash t : \tau}{\Delta; \Gamma \vdash \Lambda_{\alpha\leqslant:\sigma}.t\ : \forall_{\alpha\leqslant:\sigma}.\tau}\ \text{Type-Abst} \qquad \frac{\Delta; \Gamma \vdash t : \forall_{\alpha\leqslant:\sigma}.\tau \qquad \Gamma \vdash \sigma_1\leqslant:\sigma}{\Delta; \Gamma \vdash t\ \sigma_1 : \tau[\sigma_1/\alpha]}\ \text{Type-Inst}$$

$$\frac{}{\Gamma \vdash \tau\leqslant:\tau}\ \text{Refl} \qquad \frac{}{\Gamma \vdash \tau\leqslant:\top}\ \text{Top}$$

$$\frac{\Gamma \vdash \tau_1\leqslant:\tau_2 \qquad \Gamma \vdash \tau_2\leqslant:\tau_3}{\Gamma \vdash \tau_1\leqslant:\tau_3}\ \text{Trans} \qquad \frac{}{\Gamma \vdash x \leqslant: \Gamma[x]}\ \text{Var}$$

$$\frac{\Gamma \vdash \tau_1\leqslant:\sigma_1 \qquad \Gamma \vdash \sigma_2\leqslant:\tau_2}{\Gamma \vdash \sigma_1 \to \sigma_2 \leqslant: \tau_1 \to \tau_2}\ \text{Arrow} \qquad \frac{\Gamma \vdash \tau_1\leqslant:\sigma_1 \qquad \Gamma, x\leqslant:\tau_1 \vdash \sigma_2\leqslant:\tau_2}{\Gamma \vdash \forall_{x\leqslant:\sigma_1}.\sigma_2 \leqslant: \forall_{x\leqslant:\tau_1}.\tau_2}\ \text{All}$$

Figure 5.1: Syntax, subtyping, and type checking in full System $F_{\leqslant:}$

In order to show the undecidability of subtyping only the rules of bounded quantification are necessary, therefore we concern ourselves with a minimal subtyping system encapsulating it that has a maximal type but no arrow types (Definition 5.1). However terms on this system are not particularly interesting, in fact without term variables and abstraction there are no terms at all, therefore we include the arrow types and their subtyping rule when discussing type checking in Section 5.3. The technique that allows to split the syntax as with and without arrow types is explained in Section 6.3.

The subtyping system defines the reflexive and transitive subtype relation, it is presented with named variables to improve readability.

Definition 5.1 (System $F_{\leqslant:}$) *The types and contexts are defined by the grammar*

$$\tau, \sigma, \phi : \mathsf{Type} ::= \; x \mid \forall_{x \leqslant: \sigma}.\, \tau \mid \top \qquad \Gamma : \mathsf{Context} ::= [\,] \mid \Gamma, x \leqslant: \tau$$

$\_ \vdash \_ \leqslant: \_ : \mathsf{Context} \to \mathsf{Type} \to \mathsf{Type} \to \mathbb{P}$ *is the three place relation with constructors corresponding to the following rules:*

$$\frac{}{\Gamma \vdash \tau \leqslant: \tau}\ \textit{Refl} \qquad \frac{}{\Gamma \vdash \tau \leqslant: \top}\ \textit{Top}$$

$$\frac{\Gamma \vdash \tau_1 \leqslant: \tau_2 \qquad \Gamma \vdash \tau_2 \leqslant: \tau_3}{\Gamma \vdash \tau_1 \leqslant: \tau_3}\ \textit{Trans} \qquad \frac{}{\Gamma \vdash x \leqslant: \Gamma[x]}\ \textit{Var}$$

$$\frac{\Gamma \vdash \tau_1 \leqslant: \sigma_1 \qquad \Gamma, x \leqslant: \tau_1 \vdash \sigma_2 \leqslant: \tau_2}{\Gamma \vdash \forall_{x \leqslant: \sigma_1}.\sigma_2 \leqslant: \forall_{x \leqslant: \tau_1}.\tau_2}\ \textit{All}$$

where $\Gamma[x] = \tau \iff x \leqslant: \tau \in \Gamma$.

Pierce shows the undecidability of subtyping in System $F_{\leqslant:}$ (Theorem 5.18) by reduction from the halting problem of two-counter machines. He defines an intermediate machine model, the rowing machines (Section 4.2), then gives a definition of two-counter machines (Section 4.1) and shows that a two-counter machine M can be encoded as a rowing machine $\mathcal{R}(M)$ so that M halts if and only if $\mathcal{R}(M)$ halts (Theorem 4.13). In order to show that the halting problem of rowing machines reduces to subtyping, Pierce shows that for any rowing machine M:

$$M \text{ halts} \iff \vdash \sigma \leqslant: \mathcal{F}(M)$$

where σ is a closed type independent of M and $\mathcal{F}(M)$ is the encoding of the machine as a type. Induction alone does not go through as a number of invariants are required:

($\Rightarrow$) By induction on the trace of M, in order to encode the stepping of the machine we need:

- The **flip property** that allows us to *flip* inequalities, this enables the rebounding of the body of a quantifier in the right hand side by exploiting contravariance. The flip operator and its property are defined as follows:

$$\overline{\tau} := \forall_{x \leqslant: \tau}.x \qquad \Gamma \vdash \overline{\sigma} \leqslant: \overline{\tau} \iff \Gamma \vdash \tau \leqslant: \sigma$$

  The flip property already is a theorem in the system, however it is mentioned here because it is one of the features that will be refinements.

- The **eager substitution** of variables, to match the handling of registers by the rowing machine:

$$x \leqslant: \phi \vdash \sigma \leqslant: \tau \iff \vdash \sigma[\phi/x] \leqslant: \tau[\phi/x]$$

  This property says that types may be overaproximated, that is, variables may be replaced by their upper bounds. Of course this does not hold in general; consider $\phi = \sigma = \top$ and $\tau = x$, then the right side is derivable but the left side is not. The *polarized* syntax (See Def. 5.6) forbids such counterexamples.

($\Leftarrow$) By induction on the derivation we require:

- **Restricted transitivity**, if the last rule of the derivation is *Trans* then the intermediate type is arbitrary because it does not appear in the conclusion. This is problematic because in order to apply the inductive hypothesis the intermediate type must have the form of a translated rowing machine.

- The registers of the rowing machine are encoded as quantifiers so if M consists of w registers then $\mathcal{F}(M)$ has a prefix of w quantifiers. Therefore if the last rule of the derivation is *All* then the preceding $w - 1$ rules should also be *All*. This is enforced by **polyadic binders**, that is, quantifiers that bind w variables independent of each other.

5.1 Variants of System $F_{\leqslant:}$

Pierce named the intermediate systems based on properties of the **naive search algorithm** for derivations that recursively tries applicable rules and may loop. The algorithm is only used informally to give an intuition on how similar the computational model of the system is to the rowing machines, therefore it is not introduced.

The intermediate systems address the requirements and are defined incrementally so that the first system is the most similar to System $F_{\leqslant:}$ and the last system has all the required invariants so it is closer to the machine abstraction.

$F^N_{\leqslant:}$ Restricts transitivity. In this system the naive search algorithm can be thought as producing, if any, the **normal** forms of derivations by choosing the rules for $\top$ and reflexivity for variables whenever possible.

$F^D_{\leqslant:}$ Introduces the syntax with polyadic binders and incorporates the flip property into the rules. Additionally this system already has eager substitution as a theorem. In this system the rules have at most one premise so the derivations are linear and the naive search algorithm is a **deterministic** procedure.

$F^F_{\leqslant:}$ Incorporates eager substitution into the rules, allowing a smooth transition to the rowing machines. The naive search algorithm of this system preemptively instantiates variables, so the derivations keep the context empty and may be considered **flattened**.

5.1.1 Syntax-directed Subtyping

The first variant uses the same syntax as the full system, it restricts the rules that have arbitrary types in the conclusion so that each subtyping statement is the result of at most one rule.

Definition 5.2 (System $F^N_{\leqslant:}$) *The normal system is the three place relation* $\_ \vdash_N \_ \leqslant: \_ : \mathsf{Context} \to \mathsf{Type} \to \mathsf{Type} \to \mathbb{P}$ *with constructors corresponding to the following rules*

$$\frac{}{\Gamma \vdash_N x \leqslant: x}\ \mathit{NRefl} \qquad \frac{}{\Gamma \vdash_N \tau \leqslant: \top}\ \mathit{NTop} \qquad \frac{\Gamma \vdash_N \Gamma\ x \leqslant: \tau}{\Gamma \vdash_N x \leqslant: \tau}\ \mathit{NVar}$$

$$\frac{\Gamma \vdash_N \tau_1 \leqslant: \sigma_1 \qquad \Gamma, x \leqslant: \tau_1 \vdash_N \sigma_2 \leqslant: \tau_2}{\Gamma \vdash_N \forall_{x \leqslant: \sigma_1}.\sigma_2 \leqslant: \forall_{x \leqslant: \tau_1}.\tau_2}\ \mathit{NAll}$$

The new variable rule has a general type as the right hand side of the conclusion that is identical in the premise, this rule can also be read as an instance of transitivity between the variable, its bound in the context, and the general type. The new reflexivity rule applies only to variables and the transitivity rule is dropped, however the system is still reflexive and transitive in general so the reduction is immediate.

Lemma 5.3 *For all* Γ *and* τ*:* $\quad \Gamma \vdash_N \tau \leqslant: \tau$.

Lemma 5.4 *For all* Γ, σ, τ, *and* υ*: if* $\Gamma \vdash_N \sigma \leqslant: \tau$ *and* $\Gamma \vdash_N \tau \leqslant: \upsilon$ *then* $\Gamma \vdash_N \sigma \leqslant: \upsilon$.

Theorem 5.5 *For all* Γ, σ, *and* τ*:* $\quad \Gamma \vdash_N \sigma \leqslant: \tau \iff \Gamma \vdash \sigma \leqslant: \tau$

5.1.2 Polarized Syntax

To define the next system we need a new syntax for types that classifies them by whether they are allowed to be used to the left (negative) or right (positive) of the subtyping relation. Additionally the syntax uses the polyadic binders for positive quantifiers, and only unbounded negative quantifiers so one of the premises of the quantifier rule is always trivial. The bodies of both positive and negative quantifiers are flipped types, it is easy to check that the operator indeed *flips* the signs of polarized types.

Definition 5.6 (Polarized syntax) *For a natural number w, the positive and negative types are defined by the following grammar:*

$$\tau^+ : \mathsf{pType}_w^+ ::= \top \mid \forall_{x_1 \leqslant: \tau_1^-, \ldots, x_w \leqslant: \tau_w^-}. \overline{\tau^-}$$

$$\tau^- : \mathsf{pType}_w^- ::= x_i \mid \forall_{x_1 \leqslant: \top, \ldots, x_w \leqslant: \top}. \overline{\tau^+} \qquad i \leq w$$

The previous counter example for eager substitution is not well formed in the polarized syntax because variables are only allowed at the left hand side while $\top$ is only allowed at the right, in fact eager substitution is a theorem in the following system.

The deterministic system is defined with the height of the derivation as an index so it is incremented appropriately. To avoid notational clutter the signs of types and the bounds of negative quantifiers are omitted. The natural number w is also omitted as it is a constant in this section and the next, it only becomes relevant when encoding rowing machines in section 4.2.

Definition 5.7 (System $F_{\leqslant:}^D$) *The deterministic system is the four place relation $\_ \vdash_D^- \_ \leqslant: \_ : \mathbb{L}(\mathsf{pType}^-) \to \mathbb{N} \to \mathsf{pType}^- \to \mathsf{pType}^+ \to \mathbb{P}$ with constructors corresponding to the following rules:*

$$\frac{}{\Gamma \vdash_D^0 \tau \leqslant: \top}\ \textit{DTop}$$

$$\frac{\Gamma \vdash_D^n \Gamma\ x_i \leqslant: \forall_{y_1 \leqslant: \phi_1, \ldots, y_w \leqslant: \phi_w}.\ \overline{\tau}}{\Gamma \vdash_D^{\mathsf{S}n} x_i \leqslant: \forall_{y_1 \leqslant: \phi_1, \ldots, y_w \leqslant: \phi_w}.\ \overline{\tau}}\ \textit{DVar}$$

$$\frac{\Gamma, x_1 \leqslant: \phi_1, \ldots, x_w \leqslant: \phi_w \vdash_D^n \tau \leqslant: \sigma}{\Gamma \vdash_D^{\mathsf{S}n} \forall_{x_1, \ldots, x_w}. \overline{\sigma} \leqslant: \forall_{x_1 \leqslant: \phi_1, \ldots, x_w \leqslant: \phi_w}.\ \overline{\tau}}\ \textit{DAllFlip}$$

Reflexivity is no longer well formed so the rule is dropped. The new rule for variables is specialized so the right hand side is not $\top$, this corresponds to the algorithm

that builds derivations choosing the rule for $\top$ whenever possible. The new quantifier rule incorporates the flip property so it removes the flip operator of the bodies in the conclusion by having the bodies switch side in the premise.

In order to show the reduction from this system, we need to define the encoding of polarized types as regular types. The encoding is straightforward in the named presentations, however the deBrujin indices complicate the formalization, the details are explained in Chapter 3.

Definition 5.8 *We use the same notation for three functions; the encoding of positive types* $[\![-]\!] : \mathsf{pType}^+ \to \mathsf{Type}$*, of negative types* $[\![-]\!] : \mathsf{pType}^- \to \mathsf{Type}$*, and of contexts:*

$$
\begin{aligned}
[\![\top]\!] &:= \top \\
[\![\forall_{x_1 \leqslant: \phi_1, \ldots, x_w \leqslant: \phi_w}.\tau]\!] &:= \forall_{x_1 \leqslant: [\![\phi_1]\!]} \ldots \forall_{x_1 \leqslant: [\![\phi_1]\!]}.[\![\tau]\!] \\
[\![x_i]\!] &:= x_i \\
[\![\forall_{x_1, \ldots, x_w}.\tau]\!] &:= \forall_{x_1 \leqslant: \top} \ldots \forall_{x_w \leqslant: \top}.[\![\tau]\!] \\
[\![[]]\!] &:= [] \\
[\![\Gamma, x_i \leqslant: \tau]\!] &:= [\![\Gamma]\!], x_i \leqslant: [\![\tau]\!]
\end{aligned}
$$

The *DAllFlip* rule corresponds to w uses of *NAll* and *NTop* followed by the flip property, therefore when analyzing a derivation of the normal system it is not enough to consider the premise of the last rule but of several rules before. At this point a height-indexed reformulation of System $F^N_{\leqslant:}$ is required. The height-indexed variant is not used as the original definition as it unnecessarily complicates previous proofs.

Theorem 5.9 *For all* Γ*,* σ*, and* τ*:* $\quad (\exists n.\ \Gamma \vdash^n_D \sigma \leqslant: \tau) \iff [\![\Gamma]\!] \vdash_N [\![\sigma]\!] \leqslant: [\![\tau]\!]$

Proof. ($\Rightarrow$) By induction on the derivation.
($\Leftarrow$) By complete induction on the height of the derivation. □

5.1.3 Eager Substitution

We can show a generalization of eager substitution; instead of considering a singleton context the lemma applies to any nonempty context, and the variables to be substituted are those that were first stored. Additionally the derivation produced has smaller or equal height so it can be used when doing complete induction on the height of derivations.

Lemma 5.10 *For all closed* ϕ_1*,...,* ϕ_w*, and* n *if* $\quad x_1 \leqslant: \phi_1, \ldots, x_w \leqslant: \phi_w, \Gamma \vdash^n_D \sigma \leqslant: \tau$
then there is an m *such that* $m \leq n$ *and*
$\Gamma[\phi_1/x_1, \ldots, \phi_w/x_w] \vdash^m_D \sigma[\phi_1/x_1, \ldots, \phi_w/x_w] \leqslant: \tau[\phi_1/x_1, \ldots, \phi_w/x_w]$

Proof. By induction on the given derivation. □

To complete the proof of eager substitution we need the backwards implication, however this lemma is used when doing induction on the derivation itself so the height is left unspecified.

Lemma 5.11 *For all closed ϕ_1,..., ϕ_w*
if $(\exists n.\ \Gamma[\phi_1/x_1, \ldots, \phi_w/x_w] \vdash_D^n \sigma[\phi_1/x_1, \ldots, \phi_w/x_w] \leqslant: \tau[\phi_1/x_1, \ldots, \phi_w/x_w])$
then $(\exists n.\ x_1 \leqslant: \phi_1, \ldots, x_w \leqslant: \phi_w, \Gamma \vdash_D^n \sigma \leqslant: \tau)$

Proof. By induction on the given derivation. □

The final refinement incorporates eager substitution on the quantifier rule.

Definition 5.12 (System $F^F_{\leqslant:}$) *The flattened system is the three place relation $\vdash_F^- \_ \leqslant: \_ : \mathbb{N} \to \mathsf{pType}^- \to \mathsf{pType}^+ \to \mathbb{P}$ with constructors corresponding to the following rules.*

$$\frac{}{\vdash_F^0 \tau \leqslant: \top}\ \mathit{FTop}$$

$$\frac{\vdash_F^n \tau[\phi_1/x_1, \ldots, \phi_w/x_w] \leqslant: \sigma[\phi_1/x_1, \ldots, \phi_w/x_w]}{\vdash_F^{\mathsf{S}n} \forall_{x_1,\ldots,x_w}.\overline{\sigma} \leqslant: \forall_{x_1 \leqslant: \phi_1, \ldots, x_w \leqslant: \phi_w}.\ \overline{\tau}}\ \mathit{FAllFlip}$$

The rule for variables is dropped and contexts are omitted, as variables are immediately substituted by their bounds so they need not be stored and the context is always empty. The new quantifier rule skips the instances of the old rule for variables so when analyzing a derivation of the deterministic system once again it is not enough to consider the premise of the last rule so complete induction on the height is necessary.

Theorem 5.13 *For all closed σ and τ:* $(\exists n.\ \vdash_F^n \sigma \leqslant: \tau) \iff (\exists m.\ \vdash_D^m \sigma \leqslant: \tau)$

Proof. ($\Rightarrow$) By induction on the derivation using Lemma 5.11 for the quantifier case.
($\Leftarrow$) By complete induction on the height of the given derivation and case analysis of the last rule, using Lemma 5.10 for the quantifier case. □

5.2 Undecidability of Subtyping

We can encode rowing machines (Definition 4.9) as subtyping problems in an appropriate way; if the rowing machine has `Halt` as its first row then it it should be encoded as a statement that reaches a sub-problem with $\top$ on the right, alternatively

if the rowing machine steps into another then its encoding should be a judgment that reaches the encoding of the the second machine.

We write $\mathcal{F}$ for both the encoding of rowing machines and rows, they are functions between different types so there is no ambiguity. The encoding of a row of width w is a negative type of width $w+1$, the first bound is reserved for a closed type that encodes flipping and rebounding. Meanwhile the encoding of a rowing machine of width w is a positive type of width $w+1$.

Definition 5.14 *For all $w > 0$ the encoding $\mathcal{F} : \mathsf{Row}_w \to \mathsf{pType}^-_{w+1}$ is defined as:*

$$\begin{aligned}
\mathcal{F}(x_i) &:= x_i \\
\mathcal{F}(\texttt{Halt}) &:= \forall_{x_0,\ldots,x_w}.\,\overline{\top} \\
\mathcal{F}(\lambda_{x_1,\ldots,x_w}.\langle \rho_1, \ldots, \rho_w \rangle) &:= \forall_{x_0,\ldots,x_w}.\overline{\forall_{y_0 \leqslant: x_0, y_1 \leqslant: \mathcal{F}(\rho_1), \ldots, y_w \leqslant: \mathcal{F}(\rho_w)}.\overline{\mathcal{F}(\rho_1)}}
\end{aligned}$$

The encoding of a rowing machine $\mathcal{F} : \mathsf{RM}_w \to \mathsf{pType}^+_{w+1}$ is defined as follows:

$$\mathcal{F}(\langle \rho_1 \ldots \rho_w \rangle) := \forall_{x_0 \leqslant: \sigma_w, x_1 \leqslant: \mathcal{F}(\rho_1), \ldots, x_w \leqslant: \mathcal{F}(\rho_w)}.\overline{\mathcal{F}(\rho_1)}$$

where $\sigma_w := \forall_{x_0,\ldots,x_w}.\overline{\forall_{y_0 \leqslant: x_0, \ldots, y_w \leqslant: x_w}.\overline{x_0}}$.

The closed type σ_w (for the given w that is the number of registers) is the type mentioned at the beginning of the chapter and serves as the left hand side of the subtyping statements that check translations of rowing machines. In the following lemma both implications are used in the proof of reduction.

Lemma 5.15 *For all rowing machines of width w M, M', and $n : \mathbb{N}$, if $M \succ_R M'$ then $\vdash_F^{n+2} \sigma_w \leqslant: \mathcal{F}(M) \iff \vdash_F^n \sigma_w \leqslant: \mathcal{F}(M')$*

Proof. Assuming $M \succ_R M'$.
($\Rightarrow$) By induction on the given derivation.
($\Leftarrow$) By two applications of the *FAllFlip* rule. □

Lemma 5.16 *For all ρ_2,..., ρ_w* $\quad \vdash_F^2 \sigma_w \leqslant: \mathcal{F}(\langle \texttt{Halt}, \rho_2 \ldots \rho_w \rangle)$

Theorem 5.17 *A rowing machine of width w M halts if and only if there is an n such that $\vdash_F^n \sigma_w \leqslant: \mathcal{F}(M)$.*

Proof. ($\Rightarrow$) By induction on the reflexive transitive closure of the step relation.
($\Leftarrow$) By complete induction on the height of the derivation with the rowing machine quantified. □

Theorem 5.18 *$F_{\leqslant:}$ subtyping is undecidable.*

Proof. From the undecidability of RM halting (Theorem 4.14), by a chain of many-reductions:

$$\begin{aligned}
\text{RM halting} &\preceq_m F^F_{\leqslant:} \text{ subtyping} && \text{by Theorem 5.17}\\
&\preceq_m F^D_{\leqslant:} \text{ subtyping} && \text{by Theorem 5.13}\\
&\preceq_m F^N_{\leqslant:} \text{ subtyping} && \text{by Theorem 5.9}\\
&\preceq_m F_{\leqslant:} \text{ subtyping} && \text{by Theorem 5.5}
\end{aligned}$$

□

5.3 Undecidability of Type Checking

In order to show that type checking is undecidable Pierce constructs a term that is well-typed if and only if a given subtyping statement holds. We define a different term that simplifies the argument, however there are two complications. First, the subsumption rule is always applicable and introduces an intermediate unknown type so we need to argue about the height of such derivations as (non-reflexive) subsumptions cannot continue forever. Second, the subtyping assumptions are statements of the Full System $F_{\leqslant:}$ (Def. 5.1) and therefore might be the result of the transitivity rule, which complicates the analysis as we discussed earlier on the chapter. Therefore we need a variant of the type checking system that addresses these difficulties.

Definition 5.19 (Height-indexed type checking)

$$\frac{x : \tau \in \Delta}{\Delta;\Gamma \vdash^0 x : \tau}\ \textit{Var} \qquad \frac{\Delta;\Gamma \vdash^n t : \sigma \qquad \Gamma \vdash \sigma \leqslant: \tau}{\Delta;\Gamma \vdash^{\mathsf{S}n} t : \tau}\ \textit{Subsumption}$$

$$\frac{\Delta, x : \sigma;\Gamma \vdash^n t : \tau}{\Delta;\Gamma \vdash^{\mathsf{S}n} \lambda_{x:\sigma}.t\ : \sigma \to \tau}\ \textit{Term-Abst} \qquad \frac{\Delta;\Gamma \vdash^n t : \sigma \to \tau \qquad \Delta;\Gamma \vdash^m u : \sigma}{\Delta;\Gamma \vdash^{\mathsf{S}(\mathsf{max}\ n\ m)} t\ u : \tau}\ \textit{Term-Inst}$$

$$\frac{\Delta;\Gamma, \alpha \leqslant: \sigma \vdash^n t : \tau}{\Delta;\Gamma \vdash^{\mathsf{S}n} \Lambda_{\alpha\leqslant:\sigma}.t\ : \forall_{\alpha\leqslant:\sigma}.\tau}\ \textit{Type-Abst} \qquad \frac{\Delta;\Gamma \vdash^n t : \forall_{\alpha\leqslant:\sigma}.\tau \qquad \Gamma \vdash \sigma_1 \leqslant: \sigma}{\Delta;\Gamma \vdash^{\mathsf{S}n} t\ \sigma_1 : \tau[\sigma_1/\alpha]}\ \textit{Type-Inst}$$

We can show that a given subtyping statement holds if and only if a certain term type checks.

Lemma 5.20 *For all Γ and types σ and τ:*

$$\Gamma \vdash \sigma \leqslant: \tau \iff \exists n.\ [\,];\Gamma \vdash^n (\Lambda_{\alpha\leqslant:\tau}.\lambda_{x:\alpha}.x)\ \sigma\ : \sigma \to \sigma$$

Proof. Throughout the proof we change each $F_{\leqslant:}$ subtyping statement with a $F^N_{\leqslant:}$ subtyping statement using Theorem 5.5.

($\Rightarrow$) By *Type-Inst* followed by *Type-Abst* and *Term-Abst*.
($\Leftarrow$) Inverting the rules we have two cases, either the derivation comes from the corresponding rule as in the forward case, or it comes from subsumption in which case we do induction on the height of the derivation to arrive at the appropriate rule. □

Theorem 5.21 *$F_{\leqslant:}$ type checking is undecidable.*

Proof. From the undecidability of $F_{\leqslant:}$ subtyping, by the previous lemma we have:

$$F_{\leqslant:} \text{ subtyping } \preceq_m F_{\leqslant:} \text{ type checking}$$

□

Chapter 6

Mechanization Artifacts

In this chapter we review some techniques used in the mechanized proof that are unrelated to Pierce's proof. We focus on design choices that might be interesting to the reader, as there are plenty that were made out of convenience (like utility functions) and can be changed without much deliberation. We also leave out the techniques that are necessitated by the external tools used, for example those regarding the de Brujin presentation given by Autosubst 2, and the synthetic undecidability lemmas provided by the Undecidability Library.

Specifically, we study stronger induction principles than the ones generated automatically by Coq for certain data types. Additionally we give a technique that enables to unfold instances of these induction principles applied to constructor-headed terms. Finally, we present a technique for modular syntax by means of a boolean flag that is passed implicitly most of the time and is only set to a specific value when the relevant syntactic construct is allowed to appear or not.

6.1 Custom Induction Principles

The types corresponding to the polarized syntax of some variants of System $F_{\leqslant:}$ (see 5.6) as well as the syntax of rows of the rowing machines (see 4.9) contain a constructor which takes a fixed number of arguments of the same type, in fact because this number is fixed throughout the proof it is natural to implement the syntax with a vector container. As an example of this section we will use the polarized syntax. Recall that the syntax is defined as follows.

Definition 6.1 (Polarized syntax) *For a natural number w, the positive and negative types are defined by the following grammar:*

$$\tau^+ : \mathsf{pType}^+_w ::= \top \mid \forall_{x_1 \leqslant: \tau_1^-, \ldots, x_w \leqslant: \tau_w^-}. \overline{\tau^-}$$

$$\tau^- : \mathsf{pType}^-_w ::= x_i \mid \forall_{x_1 \leqslant: \top, \ldots, x_w \leqslant: \top}. \overline{\tau^+} \qquad i \leq w$$

```
Inductive ptype (n :nat) : Type :=
| top : ptype n
| bAllN : Vector.t (ntype n) w → ntype (S n) → ptype n
with ntype (n : nat) : Type :=
| var_ntype : fin w → fin n → ntype n
| uAllN : ptyspe (S n) → ntype n.
```

Figure 6.1: Mutual inductive types implementing the polarized syntax. Note that only the positive quantifier constructor uses vectors as negative quantifiers are always unbounded. The natural number `w` is a variable of the section.

In order to implement such syntax we need a pair of mutually inductive types; the quantifiers of both positive and negative types have a body of the opposite sign. Additionally the syntax is implemented as well-scoped (see Section 3.1) for reasons discussed in Chapter 5. Therefore the types have a natural number index that is incremented below quantifiers and variables consist of a pair of finite elements; one specifying the binder and the other the bound inside the vector in that binder. However having a vector argument in one constructor makes the induction principle generated automatically by Coq insufficient. In particular, the quantifier hypothesis says nothing about the elements in the vector.

```
Fact ptype_ind : ∀ (n : nat) (P : ptype n → Prop),
  P (top n) →
  (∀ (t : Vector.t (ntype n) w) (n : ntype (S n)), P (bAllN n t n)) →
  ∀ p : ptype n, P p

Fact ntype_ind : ∀ (n : nat) (P : ntype n → Prop),
  (∀ (f : fin w) (f0 : fin n), P (var_ntype n f f0)) →
  (∀ p : ptype (S n), P (uAllN n p)) →
  ∀ n : ntype n, P n
```

Figure 6.2: Generated inductive principles.

We require an induction principle that combines both therefore it needs two predicates, one for positive types `P`, and one for negative types `Q`. Additionally, the positive quantifier hypothesis of the new principle needs to traverse the vector container. The mutual induction principle is given in Fig. 6.3. To show this stronger induction principle we use a mutual size induction that can show the conjunction of the two predicates. It needs to show `P` (respectively `Q`) by showing `Q` (resp. `P`) for smaller types. See Fig. 6.4.

```
Fact ptype_ntype_mutind :
∀ (P : ∀ n : nat, ptype n → Prop) (Q : ∀ n : nat, ntype n → Prop),
  (∀ (n : nat), P n (top n)) →
  (∀ (n : nat) (t : Vector.t (ntype n) w) (s : ntype (S n)),
    Q (S n) s → @Forall (ntype n) (Q n) _ t → P n (bAllN n t s)) →
  (∀ (n : nat) (f : fin n) (f0 : fin w), Q n (var_ntype n f f0)) →
  (∀ (n : nat) (p : ptype (S n)), P (S n) p → Q n (uAllN n p)) →
  ∀ n : nat, (∀ p : ptype n, P n p) ∧ (∀ n : ntype n, Q n n).
```

Figure 6.3: Custom mutually inductive principle. To give even more flexibility the predicates are generalized to any scope.

```
Fact ptype_ntype_size_ind :
∀ (P : ∀ n, ptype n → Prop) (Q : ∀ n, ntype n → Prop),
  (∀ n t, (∀ n' s, nsize s < psize t → Q n' s) → P n t) →
  (∀ n t, (∀ n' s, psize s < nsize t → P n' s) → Q n t) →
  ∀ n : nat, (∀ t, P n t) ∧ (∀ s, Q n s).
```

Figure 6.4: Mutual size induction. It uses the size functions `psize` and `nsize`, which count the number symbols.

However there is now a complication; applying the mutual inductive principle to a constructor-headed term will not simplify to a useful term in general, because the inductive principle is defined by size induction. We will explore how to handle this issue in the next section.

6.2 Size Unfolding

In this section we will study how to evaluate a recursion principle defined by size recursion. The technique was suggested by Dominik Kirst for general size recursion, however we present here a concrete example. Recall the definition of the syntax of rows for rowing machines.

Definition 6.2 (Rows) *For any $w : \mathbb{N}$ such that $w > 0$ the **rows** of width w are defined by the following grammar:*

$$\rho : \mathsf{Row}_w ::= x_i \mid \lambda_{x_1,\ldots,x_w}.\langle \rho_1, \ldots, \rho_w \rangle \mid \texttt{Halt} \qquad i \leq w$$

Once again the implementation uses a vector container for the body of the abstract rows. The syntax is also well-scoped so the variables are again a pair of finite types. The recursion principle generated automatically by Coq is again insufficient, it does not traverse the vector argument.

```
Inductive row (n : nat) : Type :=
| var_row : fin w → fin n → row n
| abst : Vector.t (row (S n)) w → row n
| halt : row n.
```

Figure 6.5: Implementation of the row syntax. The natural number `w` is a variable of the section.

```
Fact row_rec : ∀ (n : nat) (P : row n → Type),
  (∀ i j, P (var_row i j)) →
  (∀ v, P (abst v)) →
  P halt →
  ∀ r : row n, P r
```

Figure 6.6: Generated recursion principle.

```
Fact row_rec : ∀ (w : nat) (P : ∀ n, @row w n → Type),
  (∀ n i j, P n (var_row i j)) →
  (∀ n v, (∀ i, P (S n) v[i]) → P n (abst v)) →
  (∀ n, P n halt) →
  ∀ n r, P n r.
```

Figure 6.7: Custom recursion principle.

The custom recursion principle is defined by straightforward size induction. Therefore we have the issue that evaluating the principle will not produce useful terms in general. We define the function in Fig. 6.8 to unfold the recursion and set the recursion principle as opaque. To show the unfolding of the recursion principle we need the general unfolding of size recursion which is given in Fig. 6.9.

```
Fact row_rec_unfold {P w Hv Ha Hh n r} :
  row_rec P w Hv Ha Hh n r = match r with
    | var_row i j ⇒ Hv n i j
    | abst v ⇒ Ha n v (fun i ⇒ row_rec P w Hv Ha Hh (S n) v[i])
    | halt ⇒ Hh n end.
```

Figure 6.8: Unfolding of the recursion principle.

```
Fact row_size_rec (P : ∀ w m, @row w m → Type) : ∀ w m,
  (∀ m x, (∀ m y, size y < size x → P w m y) → P w m x) →
  ∀ x, P w m x.
Fact row_size_rec_unfold {P w F m r} :
  row_size_rec P w m F r = F m r (fun m s H ⇒ row_size_rec P w m F s).
```

Figure 6.9: General unfolding of size recursion. The proof of both of these facts is by indution on `size t` with functional extensionality.

6.3 Modularized Syntax

There are several techniques for modular syntax supported by Coq, in particular Coq à La Carte [19] offers a general purpose approach compatible with Autosubst 2. However, as the code was originally developed without modular syntax porting it to the Coq à La Carte approach would involve rewriting most of it. We chose a technique for modular syntax that allows to reuse as much code as possible, inspired by the technique used in the undecidability results of First Order Logic [23] in the Undecidability Library, where formulas are considered with or without falsity by means of a boolean flag.

Recall that in order to show that subtyping is undecidable the only constructions needed are bounded quantification and the type $\top$ (See Fig. 6.10), however without term abstraction such a type system has no terms at all. Therefore in order to show the undecidability of type checking we have to consider arrow types as well (See Fig. 6.11).

In order to give a single implementation for both syntax with and without arrow types we give a definition with a boolean flag that the constructors pass accordingly (See Fig. 6.12). The proofs about the systems with and without arrows are mostly identical except for the extra case. Instead of duplicating proofs we define a general system with the boolean flag as a variable and show the proofs in this system, later instantiating the flag variable to obtain the specific results. We generalize the Normal System (Definition 5.2) as it shares the same syntax (See Fig. 6.13).

```
Inductive type^↛ : Type :=
| var_type^↛ : ℕ → type^↛
| top : type^↛
| all : type^↛ → type^↛ → type^↛.

Inductive sub^↛ (Γ : list type^↛) : type^↛ → type^↛ → Prop :=
| Refl' τ : Γ ⊢^↛ τ <: τ
| Trans' σ τ υ : Γ ⊢^↛ σ <: τ → Γ ⊢^↛ τ <: υ → Γ ⊢^↛ σ <: υ
| Top' τ : Γ ⊢^↛ τ <: top
| Var' n : Γ ⊢^↛ var_type^↛ n <: nth_default (var_type^↛ n) Γ n
| All' σ1 σ2 τ1 τ2 : Γ ⊢^↛ τ1 <: σ1 → map ⟨↑⟩ (τ1 :: Γ) ⊢^↛ σ2 <: τ2 →
    Γ ⊢^↛ (all σ1 σ2) <: (all τ1 τ2)
where "Γ ⊢^↛ σ <: τ" := (sub^↛ Γ σ τ).
```

Figure 6.10: Subtyping without arrows.

```
Inductive type^→ : Type :=
| var_type^→ : ℕ → type^→
| top : type^→
| arr : type^→ → type^→ → type^→
| all : type^→ → type^→ → type^→.

Inductive sub^→ (Γ : list type^→) : type^→ → type^→ → Prop :=
| Refl τ : Γ ⊢^→ τ <: τ
| Trans σ τ υ : Γ ⊢^→ σ <: τ → Γ ⊢^→ τ <: υ → Γ ⊢^→ σ <: υ
| Top τ : Γ ⊢^→ τ <: top
| Var n : Γ ⊢^→ var_type^→ n <: nth_default (var_type^→ n) Γ n
| Arr σ1 σ2 τ1 τ2 : Γ ⊢^→ τ1 <: σ1 → Γ ⊢^→ σ2 <: τ2 → Γ ⊢^→ (arr σ1 σ2) <: (arr τ1 τ2)
| All σ1 σ2 τ1 τ2 : Γ ⊢^→ τ1 <: σ1 → map ⟨↑⟩ (τ1 :: Γ) ⊢^→ σ2 <: τ2 →
    Γ ⊢^→ (all σ1 σ2) <: (all τ1 τ2)
where "Γ ⊢^→ σ <: τ" := (sub^→ Γ σ τ).
```

Figure 6.11: Subtyping with arrows.

```
Inductive arrow_flag := arrow_off | arrow_on.

Inductive type : arrow_flag → Type :=
| var_type {b} : ℕ → type b
| top {b} : type b
| arr : type arrow_on → type arrow_on → type arrow_on
| all {b} : type b → type b → type b.
```

Figure 6.12: Unscoped syntax with boolean flag.

```
Inductive subN : ∀ (b:arrow_flag), list (type b)→ type b → type b → Prop :=
| NRefl {b: arrow_flag} Γ n : Γ ⊢N var_type n <: var_type n
| NVar {b: arrow_flag} Γ σ n : Γ ⊢N nth_default (var_type n) Γ n <: σ →
    Γ ⊢N var_type n <: σ
| NTop {b: arrow_flag} Γ σ: Γ ⊢N σ <: top
| NArr Γ σ1 σ2 τ1 τ2: Γ ⊢N τ1 <: σ1 → Γ ⊢N σ2 <: τ2 → Γ ⊢N (arr σ1 σ2) <: (arr τ1 τ2)
| NAll {b: arrow_flag} Γ σ1 σ2 τ1 τ2: Γ ⊢N τ1 <: σ1 → map ⟨↑⟩ (τ1 :: Γ) ⊢N σ2 <: τ2 →
    Γ ⊢N (all σ1 σ2) <: (all τ1 τ2)
where "Γ ⊢N σ <: τ" := (subN _ Γ σ τ).
```

Figure 6.13: Normal subtyping with flag variable. Note how the arrow rule has no boolean variable as it is automatically instantiated to `arrow_on`.

Chapter 7

Conclusion

We give a mechanization of the proof of the undecidability of subtyping by Pierce [33] using the synthetic setting of the Coq Library of Undecidability Proofs [22]. The machine-checked presentation differs in several aspects to the textbook presentation given by Pierce.

Instead of using named variables we use several implementations of de Brujin indices [14] provided by the Autosubst 2 tool [40] for the different intermediate systems, as they have several useful properties that simplify the argument. Concretely, well-scoped syntax (See Sect. 3.1) carry an upper bound to their type variables which allows to efficiently reason about the construction that allows the substitution of the first bound introduced in a nonempty context along the rest of it (See Sect. 5.1.3). Also, polyadic binders [40] (See Sect. 3.2) bind several type variables at the same level, succinctly encoding the binders of the polarized syntax (See Sect. 5.1.2) which classifies types by whether they are allowed to the left or to the right of the subtyping relation. The combination of both these syntactic features makes the encoding of the polarized syntax nontrivial, and in particular requires the use of a bounded variant of extensionality in order to show that the encoding is well behaved (See Ch. 3). Finally, we omit one of Pierce's intermediate subtyping systems namely the *polarized* system $F^{P}_{<:}$ as the original system is not a conservative extension of it and it only serves to introduce the polarized syntax and the flip operation into the subtyping rules. Instead we introduce these features in the next system in the chain; the *deterministic* system $F^{D}_{<:}$.

Although not a goal of this work one of the presented mechanized results constitutes half of the first part of the POPLmark challenge [1], that is, the proof of transitivity of the *algorithmic* subtype relation given here as Lemma 5.4. The mathematical argument of our proof is largely inspired by the proof given by Stark [39], who also gives the rest of Part 1 of the challenge, however we focus on how the proof fits in the chain of reductions that comprises the main result of this thesis and it is therefore slightly more convoluted.

Finally, we describe some techniques that allow to write the machine-checked proof in a succinct manner although unrelated to the mathematical argument. In particular we give some inductive and recursive principles stronger than the ones produced automatically by the Coq proof assistant, in addition to a result that unfolds size recursion when applied to a constructor-headed term. Lastly, we give an implementation of syntax that allows to *turn off* specific syntactic constructors via a boolean flag.

7.1 Future Work

There are two results known to us that are consequences of Pierce's proof and are yet to be mechanized in Coq and can become future contributions to the Coq Library of Undecidability Proofs.

Wehr and Thiemann [44] use the undecidability of subtyping in the *deterministic* system (See Def. 5.7) to show the undecidability of subtyping bounded existential types, where the existentially quantified variable has either a lower or upper type bound. Their proof encodes the contravariance of bounded quantification using lower type bounds, however the proof relies on several syntactic invariants that turned out difficult to mechanize.

Hu and Lhotak [27] use the undecidability of subtyping in the *normal* system (See Def. 5.2) to show the undecidability of type checking and subtyping in the Dependant Object Types calculus. They use a similar technique to Pierce defining a variant of the type system without the general transitivity rule. Their proof is already mechanized in Agda so the Coq port should not be too complicated, it was not included in the present work due to time limitations.

Bibliography

[1] Brian Aydemir, Aaron Bohannon, Matthew Fairbairn, Nathan Foster, Benjamin Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *Lecture Notes in Computer Science*, volume 3603, pages 50–65, 2005. doi: 10.1007/11541868_4.

[2] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, apr 1991. ISSN 0956-7968. doi: 10.1017/S0956796800020025.

[3] Andrej Bauer. First Steps in Synthetic Computability Theory. *Electronic Notes in Theoretical Computer Science*, 155(1 SPEC. ISS.):5–31, may 2006. ISSN 15710661. doi: 10.1016/j.entcs.2005.11.049.

[4] Andrej Bauer. On fixed-point theorems in synthetic computability. *Tbilisi Mathematical Journal*, 10(3):167–181, jun 2017. ISSN 1875-158X. doi: 10.1515/tmj-2017-0107.

[5] Stefan Berghofer. A Solution to the PoplMark Challenge Using de Bruijn Indices in Isabelle/HOL. *Journal of Automated Reasoning 2011 49:3*, 49(3):303–326, jun 2011. ISSN 1573-0670. doi: 10.1007/S10817-011-9231-4.

[6] Richard Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999. ISSN 09567968. doi: 10.1017/S0956796899003366.

[7] Douglas Bridges and Fred Richman. Varieties of Constructive Mathematics. In *Varieties of Constructive Mathematics*. Cambridge University Press, apr 1987. ISBN 9780521318020. doi: 10.1017/CBO9780511565663.

[8] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2-3), 1988. ISSN 10902651. doi: 10.1016/0890-5401(88)90007-7.

[9] Luca Cardelli and Peter Wegner. On understanding types, data abstraction,

and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985. ISSN 15577341. doi: 10.1145/6041.6042.

[10] Arthur Charguéraud. A Solution to part 1A of the POPLmark Challenge. 2006. URL: http://www.chargueraud.org/research/2006/poplmark/solution_2/documentation.pdf.

[11] Alberto Ciaffaglione and Ivan Scagnetto. A weak HOAS approach to the POPLmark Challenge. *Electronic Proceedings in Theoretical Computer Science*, 113:109–124, mar 2013. doi: 10.4204/EPTCS.113.11.

[12] Thierry Coquand and Christine Paulin. Inductively defined types. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 417 LNCS, pages 50–66. Springer, Berlin, Heidelberg, dec 1990. ISBN 9783540523352. doi: 10.1007/3-540-52335-9_47.

[13] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in F <:. *Mathematical Structures in Computer Science*, 2(1):55–91, 1992. ISSN 0960-1295. doi: 10.1017/S0960129500001134.

[14] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. ISSN 13857258. doi: 10.1016/1385-7258(72)90034-0.

[15] Marco Dicolla. A solution of POPLMark Challenge with VCPT. 2009. URL: https://www.politesi.polimi.it/bitstream/10589/2228/3/Master_thesis.pdf.

[16] Andrej Dudenhefner. Constructive Many-One Reduction from the Halting Problem to Semi-Unification. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 216, 2022. ISBN 9783959772181. doi: 10.4230/LIPIcs.CSL.2022.18.

[17] Andrej Dudenhefner and Jakob Rehof. A simpler undecidability proof for system F inhabitation. *Leibniz International Proceedings in Informatics, LIPIcs*, 130, nov 2019. ISSN 18688969. doi: 10.4230/LIPICS.TYPES.2018.2.

[18] Yannick Forster and Fabian Kunze. A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1—-17:19. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. ISBN 978-3-95977-122-1. doi: 10.4230/LIPIcs.ITP.2019.17.

[19] Yannick Forster and Kathrin Stark. Coq à La Carte: A Practical Approach to Modular Syntax with Binders. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 186–200, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370974. doi: 10.1145/3372885.3373817.

[20] Yannick Forster, Edith Heiter, and Gert Smolka. Verification of PCP-Related Computational Reductions in Coq. *Lecture Notes in Computer Science* (*including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics*), 10895 LNCS:253–269, jul 2018. ISSN 16113349. doi: 10.1007/978-3-319-94821-8_15.

[21] Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs - CPP 2019*, pages 38–51, New York, New York, USA, jan 2019. ACM Press. ISBN 9781450362221. doi: 10.1145/3293880.3294091.

[22] Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, Maximilian Wuttke, and Dominique Larchey-wendling. A Coq Library of Undecidable Problems. *CoqPL 2020 The Sixth International Workshop on Coq for Programming Languages*, pages 1–23, 2020. URL: `https://hal.inria.fr/INRIA/hal-02944217v1`.

[23] Yannick Forster, Dominik Kirst, and Dominik Wehr. Completeness theorems for first-order logic analysed in constructive type theory: Extended version. *Journal of Logic and Computation*, 31(1):112–151, 2021. ISSN 0955-792X. doi: 10.1093/logcom/exaa073.

[24] Giorgio Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD thesis, 1990.

[25] Giorgio Ghelli. Divergence of F<: type checking. *Theoretical Computer Science*, 139(1-2):131–162, 1995. ISSN 03043975. doi: 10.1016/0304-3975(94)00037-J.

[26] Jean-Yves Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. *Thèse d'état, Université Paris VII*, 1972. URL: `https://www.cs.cmu.edu/$\sim$kw/scans/girard72thesis.pdf`.

[27] Jason Hu and Ondrej Lhoták. Undecidability of D <: and Its Decidable Fragments. *Proceedings of the ACM on Programming Languages*, 4(POPL), 2020. doi: 10.1145/3371077.

[28] Xavier Leroy. A locally nameless solution to the POPLmark challenge. Technical Report 6098, 2007. URL: https://hal.inria.fr/inria-00123945/.

[29] Barbara H. Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6): 1811–1841, 1994. ISSN 15584593. doi: 10.1145/197320.197383.

[30] Marvin Lee Minsky. *Computation: Finite and Infinite Machines.*, volume 75. apr 1968. ISBN 978-0-13-165563-8. URL: https://dl.acm.org/doi/book/10.5555/1095587.

[31] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In *All about Proofs, Proofs for All*, volume 55. College Publications, 2015. ISBN 9781848901667. URL: https://hal.inria.fr/hal-01094195.

[32] Benjamin Pierce. *Types and Programming Languages*. MIT Press. ISBN 0-262-16209-1.

[33] Benjamin Pierce. Bounded Quantification Is Undecidable. *Information and Computation*, 112(1):131–165, jul 1994. ISSN 08905401. doi: 10.1006/inco.1994.1055.

[34] Emil Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946. ISSN 0273-0979. doi: 10.1090/S0002-9904-1946-08555-9.

[35] John Reynolds. Towards a theory of type structure. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 19 LNCS, pages 408–425. Springer, Berlin, Heidelberg, 1974. ISBN 9783540068594. doi: 10.1007/3-540-06859-7_148.

[36] John Reynolds. Using category theory to design implicit conversions and generic operators. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 94 LNCS, pages 211–258. Springer, Berlin, Heidelberg, 1980. ISBN 9783540102502. doi: 10.1007/3-540-10250-7_24.

[37] Fred Richman. Church's thesis without tears. *Journal of Symbolic Logic*, 48(3): 797–803, 1983. ISSN 0022-4812. doi: 10.2307/2273473.

[38] Simon Spies and Yannick Forster. Undecidability of higher-order unification formalised in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 143–157, New York, NY, USA, jan 2020. ACM. ISBN 9781450370974. doi: 10.1145/3372885.3373832.

[39] Kathrin Stark. *Mechanising Syntax with Binders in Coq*. PhD thesis, 2019. doi: 10.22028/D291-30298.

[40] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: Reasoning with multi-sorted de Bruijn terms and vector substitutions. In *CPP 2019 - Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, Co-located with POPL 2019*, pages 166–180, New York, New York, USA, 2019. ACM Press. ISBN 9781450362221. doi: 10.1145/3293880.3294101.

[41] Aaron Stump. POPLmark 1a with Named Bound Variables. Technical report, Washington University in St. Louis, 2005. URL: `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.521.5740&rep=rep1&type=pdf`.

[42] The Coq Development Team. The Coq Proof Assistant. 2022. doi: 10.5281/ZENODO.5846982.

[43] Jérŏme Vouillon. A Solution to the PoplMark Challenge Based on de Bruijn Indices. *Journal of Automated Reasoning 2011 49:3*, 49(3):327–362, jun 2011. ISSN 1573-0670. doi: 10.1007/S10817-011-9230-5.

[44] Stefan Wehr and Peter Thiemann. On the decidability of subtyping with bounded existential types. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5904 LNCS, pages 111–127, 2009. ISBN 3642106714. doi: 10.1007/978-3-642-10672-9_10.

[45] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI: Generalized interfaces for Java. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4609 LNCS: 347–372, 2007. ISSN 16113349. doi: 10.1007/978-3-540-73589-2_17.