The Undecidability of System F Typability and Type Checking for Reductionists

Andrej Dudenhefner
Saarland University, Germany
andrej.dudenhefner@cs.uni-saarland.de

***Abstract*—The undecidability of both typability and type checking for System F (polymorphic lambda-calculus) was established by Wells in the 1990s. For type checking Wells gave an astonishingly simple reduction from semi-unification (first-order unification combined with first-order matching). For typability Wells developed an intricate calculus to control the shape of type assumptions across type derivations via term structure. This calculus of invariant type assumptions allows for a reduction from type checking to typability. Unfortunately, this approach relies on heavy machinery that complicates surveyability of the overall argument.**

The present work gives comparatively simple, direct reduction from semi-unification to System F typability. The key observation is as follows: in the existential setting of typability, it suffices to consider some specific (but not all, as for invariant type assumptions) type derivations. Additionally, the particular result requires only to consider closed types without nested quantification.

The undecidability of type checking is obtained via a folklore reduction from typability.

Profiting from its smaller footprint, correctness of the new approach is witnessed by a mechanization in the Coq proof assistant. The mechanization is incorporated into the existing Coq library of undecidability proofs. For free, the library provides constructive, mechanically verified many-one reductions from Turing machine halting to both System F typability and System F type checking.

***Index Terms*—lambda-calculus, system F, typability, type checking, undecidability, constructive mathematics, mechanization**

I. Introduction

System F [1] (also known as polymorphic λ-calculus [2]) is a typed λ-calculus which captures the notion of parametric polymorphism in functional programming. It is an integral component ($\lambda 2$) of Barendregt's λ-cube [3], and it corresponds to intuitionistic second-order propositional logic via the Curry–Howard isomorphism (for an overview see [4]).

Variants of System F form the basis of functional programming languages such as Haskell and Standard ML. Therefore, the decision problems of type checking (given a type environment Γ, a term M, and a type τ, can M be assigned τ in Γ?) and typability (given a term M, can M be assigned some type in some type environment?) are of practical relevance for programming language design. Since first asked in the 1980s [5], for (Curry-style) System F decidability of both type checking and typability was a long-standing open problem, until answered negatively in the 1990s by Wells [6], [7]. The argument is by reduction from semi-unification [8] (first-order unification combined with first-order matching) to type checking, and by reduction from type checking to typability. While the reduction from semi-unification to type checking is astonishingly simple, the same cannot be said for the reduction from type checking to typability. For this, Wells contributes an intricate calculus of *invariant type assumptions* to control the shape of types via term structure. The undecidability of typability is a simple consequence of a much stronger theorem [7, Theorem 6.14], which establishes control over types across all type derivations.

Unfortunately, the generality of Wells' argument has a downside. It requires heavy machinery to force types to bend the proverbial knee to terms. Correspondingly, the argument is challenging to survey (in the sense of Bassler [9]), teach, and mechanically verify[1]. As a result, it is natural to ask for a more *reductionistic*[2] alternative, focused exclusively on the decision problems at hand.

The present work gives a simpler argument for the undecidability of System F typability and type checking. As before, the argument is by reduction from two-inequality semi-unification (given pairs $(\sigma_1, \tau_1), (\sigma_2, \tau_2)$ of simple types, are there substitutions φ, ψ_1, ψ_2 such that $\psi_1(\varphi(\sigma_1)) = \varphi(\tau_1)$ and $\psi_2(\varphi(\sigma_2)) = \varphi(\tau_2)$?). The key observation is that for invariant type assumptions a lot of effort is required to account for all possible types and type derivations. However, in order to reduce semi-unification to typability this is not necessary. For example, it suffices to only consider type assumptions with no free variables and no nested quantification. Additionally, in the existential setting of typability there is no need to control *all* possible type derivations. This allows us to introduce a simpler tool, *prenex simulation* (Lemma 29), in order to reduce semi-unification to typability. Prenex simulation can be understood as a weaker sibling of Wells' main technical result [7, Theorem 6.14], restricted to closed types without nested quantification and tailored towards typability (cf. Definition 19 of *restricted typability*). A notable novelty of the present work is the emphasis on the type assumption $(w : \theta)$ where $\theta := \forall a. \forall b. a \to b \to b \to a$ (cf. Section III-B). It is used to build simply typed terms from terms typed by corresponding subtypes. For example, if terms M and N are assigned simple types τ and σ respectively, then the term $(w\, M\, N)$ is assigned the simple type $\sigma \to \tau$.

[1]In fact, the author tried and failed to mechanize invariant type assumptions using the Coq proof assistant.

[2]"*[Reductionism is] the practice of analysing and describing a complex phenomenon in terms of its simple or fundamental constituents, especially when this is said to provide a sufficient explanation.*" (dictionary definition)

978-1-6654-4895-6/21/$31.00 ©2021 IEEE

Finally, combined with a folklore reduction from typability to type checking, we obtain the undecidability of both.

A tangible benefit of the reductionistic approach is the feasibility to verify the results mechanically. In fact, the argument is mechanized using the Coq proof assistant and contributed to the Coq Library of Undecidability Proofs [10]. For free, the library provides constructive, mechanically verifiable evidence for many-one equivalence of System F typability and System F type checking. The contributed mechanization integrates nicely along the existing mechanized (as part of the library) undecidability result for System F inhabitation [11]. Following the design philosophy of the library, the main argument is presented as a chain of constructive many-one reductions starting from a variant of semi-unification (Problem 30). Specifically, a predicate P over the domain X (constructively) many-one reduces to a predicate Q over the domain Y, if there exists a computable function $f : X \to Y$ such that for all $x \in X$ we (constructively) have $P(x) \iff Q(f(x))$. Clearly, if P is undecidable, then so is Q.

The rest of the present work is organized as follows.

Section II: Preliminaries on System F typability (Problem 6) and type checking (Problem 7).

Section III: Restricted typability (Definition 19) and prenex simulation (Lemma 29), which constitute the technical contribution.

Section IV: Preliminaries on left-uniform, two-inequality semi-unification (Problem 34).

Section V: Many-one reduction from left-uniform, two-inequality semi-unification to System F typability (Lemma 40) using prenex simulation.

Section VI: Folklore many-one reduction from System F typability to System F type checking (Lemma 42).

Section VII: Overview over mechanized results as part of the Coq Library of Undecidability Proofs [10].

Section VIII: Concluding remarks.

II. System F

In this section we recapitulate the Curry-style System F type assignment (Definition 4) which assigns polymorphic types (Definition 2) to λ-terms (Definition 1) in a type environment (Definition 3).

Definition 1 (λ-Terms ($\mathbb{L}$))**.** The set of *λ-terms* $\mathbb{L}$, ranged over by M, N, is given by the grammar

$$M, N \in \mathbb{L} ::= x \mid (M\,N) \mid (\lambda x.M)$$

where x, y, z range over a countably infinite set of *term variables*.

For brevity, we omit superfluous parentheses where (λ) binds weakest and term application associates to the left. We abbreviate $\lambda x_1. \dots \lambda x_n.M$ by $\lambda x_1 \dots x_n.M$.

Definition 2 (Types ($\mathbb{T}$))**.** Let $\mathbb{V}$ be a countably infinite set of *type variables* ranged over by a, b, c. The set of *types* $\mathbb{T}$, ranged over by σ, τ, ρ, is given by the grammar

$$\sigma, \tau, \rho \in \mathbb{T} ::= a \mid (\sigma \to \tau) \mid (\forall a.\, \tau)$$

Again, we omit superfluous parentheses where ($\forall$) binds weakest and ($\to$) associates to the right. For example, we have $\forall a.\, a \to \forall b.\, b \to a = \Big(\forall a.\, \Big(a \to \big(\forall b.\, (b \to a)\big)\Big)\Big)$. We abbreviate $\forall a_1. \dots \forall a_n.\, \tau$ by $\forall \vec{a}.\, \tau$ where $\vec{a}$ is the sequence $a_1 \dots a_n$.

As is usual, *free* variables of a term M (resp. type τ), denoted by $\mathrm{var}(M)$ (resp. $\mathrm{var}(\tau)$), are those which occur in M (resp. τ) and are not bound by λ (resp. $\forall$). We follow the usual binder hygiene, i.e. any term (resp. type) variable is bound at most once, bound variables of two terms (resp. types) are distinct, and bound variables are distinct from free variables. Replacing occurrences of a free type variable a in σ by τ is denoted by $\sigma[a := \tau]$. The more general notion of *substitution* $\varphi : \mathbb{V} \to \mathbb{T}$ is tacitly lifted from the variable to the type domain.

We denote by $\forall.\, \sigma$ the type $\forall a_1 \dots a_n.\, \sigma$ such that $\{a_1, \dots a_n\} = \mathrm{var}(\sigma)$ and the sequence $a_1 \dots a_n$ is sorted by occurrence in σ from left to right. For example, $\forall.\, a \to b \to b \to a = \forall a\, b.\, a \to b \to b \to a$.

A *type environment*, ranged over by Γ, Δ, is a finite set of *type assumptions* of shape $(x : \sigma)$ with distinct term variables in its *domain*.

Definition 3 (Type Environment, Domain, Extension, Free Type Variables)**.**

$$\begin{aligned}
\Gamma, \Delta &::= \{x_1 : \sigma_1, \dots, x_n : \sigma_n\} \\
&\qquad \text{where } x_i \neq x_j \text{ for } 1 \leq i < j \leq n \\
\mathrm{dom}(\Gamma) &:= \{x_1, \dots, x_n\} \\
\Gamma, x : \sigma &:= \Gamma \cup \{x : \sigma\} \text{ if } x \notin \mathrm{dom}(\Gamma) \\
\mathrm{var}(\Gamma) &:= \bigcup \{\mathrm{var}(\sigma) \mid (x : \sigma) \in \Gamma\}
\end{aligned}$$

The rules of the Curry-style[3] System F with *judgements* of shape $\Gamma \vdash M : \tau$, are given by the below Definition 4 (cf. [7, Figure 4]).

Definition 4 (Curry-style System F Type Assignment)**.**

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ (\text{Var})$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\ (\text{Abs})$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (\text{App})$$

$$\frac{\Gamma \vdash M : \sigma \qquad a \notin \mathrm{var}(\Gamma)}{\Gamma \vdash M : \forall a.\, \sigma}\ (\text{Gen})$$

$$\frac{\Gamma \vdash M : \forall a.\, \sigma}{\Gamma \vdash M : \sigma[a := \tau]}\ (\text{Inst})$$

We write $\Gamma \succ M$, if we are not interested in the particular type assigned to M in Γ (Definition 5).

Definition 5 (Typability in Environment)**.** An environment Γ *types* a term M, denoted $\Gamma \succ M$, if there exists a type τ such that $\Gamma \vdash M : \tau$.

[3]In comparison, the Church-style presentation of System F includes type information on term level (cf. [4, Section 12]).

We consider the decision problems of *typability* (Problem 6) and *type checking* (Problem 7), both of which are proven undecidable by Wells [7, Theorem 6.16].

Problem 6 (Typability)**.** Given a term M, is there an environment Γ and a type τ such that $\Gamma \vdash M : \tau$?

Problem 7 (Type Checking)**.** Given an environment Γ, a term M, and a type τ, does $\Gamma \vdash M : \tau$ hold?

Quantifier-free types are called *simple* (Definition 8).

Definition 8 (Simple Types ($\mathbb{T}_\to$))**.** A type $\tau \in \mathbb{T}$ is *simple*, denoted $\tau \in \mathbb{T}_\to$, if either $\tau \in \mathbb{V}$ or $\tau = \sigma' \to \tau'$ for some simple types σ' and τ'.

For reference, the following Definition 9 and Fact 10 list common terms, types, and typings.

Definition 9 (Notable Terms and Types)**.**

$$I := \lambda x.x \qquad K := \lambda x\, y.x \qquad \omega := \lambda x.x\, x \qquad \bot := \forall a.\, a$$

Fact 10 (Notable Typings)**.**

- $\emptyset \vdash I : \forall a.\, a \to a$
- $\emptyset \vdash K : \forall a.\, a \to \forall b.\, b \to a$
- $\emptyset \vdash \omega : \bot \to \bot$

System F type derivations enjoy weakening, cut elimination for terms [4, Chapter 12] (subject reduction), and cut elimination for types [12, Theorem 3] ((Inst) before (Gen) property [7, Lemma 3.2]). This implies the following properties (Fact 11) of System F.

Fact 11 (Notable Typing Properties)**.**

1) If $\Gamma \vdash K\, M\, N : \tau$, then $\Gamma \vdash M : \tau$ and $\Gamma \succ N$.
2) If $\Gamma \vdash \lambda x.M : \sigma \to \tau$, then $\Gamma, x : \sigma \vdash M : \tau$.
3) If $\Gamma, x : \forall \vec{a}.\, \sigma \to \tau \vdash x : \sigma' \to \tau'$, then $\varphi(\sigma \to \tau) = \sigma' \to \tau'$ for some $\varphi : \mathbb{V} \to \mathbb{T}$.
4) $\Gamma \vdash M : \tau$ iff $\Gamma, x : \sigma \vdash M : \tau$ where $x \notin \mathrm{var}(M)$.

For Curry-style System F the (Gen) and (Inst) rules do not influence term structure. Therefore, a quantification $\forall a.\, \sigma$ where $a \notin \mathrm{var}(\sigma)$ is of little interest. A $\forall$I-type (Definition 12) does not contain such superfluous quantifications.

Definition 12 ($\forall$I-Type)**.** A type τ is $\forall$I, denoted $\tau \in \mathbb{T}_{\forall \mathrm{I}}$, if for every subtype $\forall a.\, \sigma$ of τ we have $a \in \mathrm{var}(\sigma)$.

Typability of a term M is equivalent to typability of M in the $\forall$I-fragment of System F by the following Lemma 13.

Lemma 13. If $\Gamma \vdash M : \tau$, then $[\Gamma]_{\forall \mathrm{I}} \vdash M : [\tau]_{\forall \mathrm{I}}$, where the function $[\cdot]_{\forall \mathrm{I}} : \mathbb{T} \to \mathbb{T}_{\forall \mathrm{I}}$ is such that

$$[a]_{\forall \mathrm{I}} = a$$
$$[\sigma \to \tau]_{\forall \mathrm{I}} = [\sigma]_{\forall \mathrm{I}} \to [\tau]_{\forall \mathrm{I}}$$
$$[\forall a.\, \sigma]_{\forall \mathrm{I}} = \begin{cases} \forall a.\, [\sigma]_{\forall \mathrm{I}} & \text{if } a \in \mathrm{var}(\sigma) \\ [\sigma]_{\forall \mathrm{I}} & \text{otherwise} \end{cases}$$

and $[\Gamma]_{\forall \mathrm{I}}$ denotes $\{x : [\sigma]_{\forall \mathrm{I}} \mid (x : \sigma) \in \Gamma\}$.

Proof. Induction on the type derivation. □

Remark 14. The converse of Lemma 13 does not hold. Consider the types $\sigma := a \to \forall b.\, a$ and $\tau := a \to a$. We have $[\sigma]_{\forall \mathrm{I}} = [\tau]_{\forall \mathrm{I}} = a \to a$, and therefore $\{x : [\sigma]_{\forall \mathrm{I}}\} \vdash x : [\tau]_{\forall \mathrm{I}}$. However, $\{x : \sigma\} \nvdash x : \tau$. Otherwise, by Fact 11.3 we would have the contradiction $\varphi(\forall b.\, a) = a$ for some $\varphi : \mathbb{V} \to \mathbb{T}$.

Relying on Lemma 13, we assume all types to be $\forall$I-types in the remainder of the present work.

Remark 15. The restriction to $\forall$I-types is strictly weaker than the restriction to *canonical type expressions* [7, Definition 3.10] in the original argument. In particular, canonical type expressions dictate the order of occurrences of bound type variables, which is more difficult to establish mechanically (cf. Section VII).

III. Prenex Simulation

In this section we establish the key *prenex simulation* (Lemma 29) in the $\forall$I-fragment of System F. Intuitively, prenex simulation allows us to fix (in the context of typability) type assumptions $(x : \forall.\, \sigma)$ where σ is simple.

Remark 16. Types $\forall.\, \sigma$ where σ is simple almost coincide with *polytypes* (or type schemes) [13, Section 3.4] in Hindley–Milner type systems. The only difference is that a polytype may refer to *basic types*, which are not present in our setting. The key similarity is that quantifiers appear only at the top level.

Using prenex simulation, it is easy (cf. [7, Section 4]) to reduce semi-unification to System F typability (Section V).

A. Invariant Type Assumption $(x : a \to a)$

Let us recapitulate the construction of terms $J_x[\cdot]$ for which every type derivation necessarily contains the type assumption $(x : a \to a)$ for some fresh type variable a. The construction of terms $J_x[\cdot]$ in the following Lemma 17 is essentially the same as of the known term J [7, Lemma 6.3][4].

Lemma 17. Let M be a term, let x be a term variable, and let

$$J_x[M] := \big(\lambda y.K\, (y\, y)\, (y\, \omega)\big)\, \big(\lambda x.K\, x\, (K\, (\lambda z.x\, (x\, z))\, M)\big)$$

We have that for all environments Γ such that $x \notin \mathrm{dom}(\Gamma)$ there is a type variable $a \notin \mathrm{var}(\Gamma)$ such that $\Gamma \succ J_x[M]$ iff $\Gamma, x : a \to a \succ M$.

Proof. Let Γ be an environment such that $x \notin \mathrm{dom}(\Gamma)$.

If $\Gamma, x : a \to a \succ M$, then it is easy to verify that $\Gamma \succ J_x[M]$ holds using Fact 10 and the type assumptions $z : \bot$ and $y : \forall a.\, (a \to a) \to a \to a$.

For the converse, assume $\Gamma \succ J_x[M]$. For some $\vec{a}$, σ, and τ we have

1) $\Gamma, x : \sigma \vdash K\, x\, (K\, (\lambda z.x\, (x\, z))\, M) : \tau$
2) $\Gamma, x : \sigma \vdash x : \tau$
3) $\Gamma, y : \forall \vec{a}.\, \sigma \to \tau \succ y\, y$
4) $\Gamma, y : \forall \vec{a}.\, \sigma \to \tau \succ y\, \omega$

[4]Historically, the discovery of the term J [7, Lemma 6.3] and its properties, which took several years of search, can be considered the key milestone in the overall undecidability proof for System F typability.

The specific type assumption $(y : \forall\vec{a}.\, \sigma \to \tau)$ in (3) and (4) is such that the type of y matches the type of the term $\lambda x.K\, x\, (K\, (\lambda z.x\, (x\, z))\, M)$ due to the top-level application.

Due to (3), the leftmost type variable in σ is some $a \notin \mathrm{var}(\Gamma)$ such that $a \in \vec{a}$. Otherwise, considering the nesting depth of $(\to)$ to the left, we could not derive some type $\sigma' \to \tau'$ for the first occurrence of y and the type σ' for the second occurrence of y. Similarly considering (4), if for ω we derive some type $\forall\vec{b}.\, (\forall\vec{c}.\, \sigma'') \to \tau''$, then leftmost type variable in σ'' is c such that $c \in \vec{c}$. Due to the subterm $x\, z$ in (1) we have $\sigma \neq a$. Overall, we have $\sigma = \forall\vec{b}.\, a \to \rho$ for some $\vec{b}, \rho$ such that $a \notin \vec{b}$.

It remains to show that $\rho = a$. Due to the subterm $x\, (x\, z)$ in (1) we have that either $\rho = a$, $\rho = b$ for some $b \in \vec{b}$, or $\rho = \bot$. The last two cases are impossible due to (2) and (3).

Overall, we have $\sigma = a \to a$, and due to (1) and Fact 11.1 we have $\Gamma, x : a \to a \succ M$. ☐

Remark 18. Since $a \notin \mathrm{var}(\Gamma)$ in Lemma 17, we can replace a by any fresh type variable.

The purpose of a type assumption $(x : a \to a)$ is twofold. First, we can establish type variable equality (used in Lemma 27) because

$$\{x : a \to a, y_1 : a_1, y_2 : a_2\} \succ K(x\, y_1)(x\, y_2) \text{ iff } a_1 = a = a_2$$

Second, we can ensure that some subtypes are type variables (used in Lemma 27 and Lemma 29) because

$$\Gamma, x : a \to a, q : \forall\vec{b}.\, \sigma \to \tau \succ q\, (x\, z) \text{ implies } \sigma \in \mathbb{V}$$

B. Restricted Typability

Similarly to *invariant type assumptions* [7, Section 5], we want to characterize typability of a term while relying on some fixed type assumption. This is made precise in the following Definition 19.

Definition 19 (Restricted Typability, $N \in M|_{(x:\sigma)}$)**.** Given a term M and a type assumption $(x : \sigma)$, the set of terms $M|_{(x:\sigma)}$ is as follows. A term N is a member of $M|_{(x:\sigma)}$ if for all environments Γ such that $x \notin \mathrm{dom}(\Gamma)$ we have $\Gamma \succ N$ iff $\Gamma, x : \sigma \succ M$.

Remark 20. By Wells' main technical result [7, Theorem 6.14] using invariant type assumptions, we know that given a term M and a type assumption $(x : \sigma)$ we can *compute* a term N such that $N \in M|_{(x:\sigma)}$. However, the particular construction requires heavy machinery which constitutes the core technical argument. In the present work we consider only assumptions of certain shape, allowing for a different, simpler approach.

A notable property of Curry-style (as opposed to Church-style) System F is the ability to generalize type assumptions without affecting the corresponding term (Lemma 21).

Lemma 21. If $\Gamma, x : \tau \succ M$ and $\{x : \sigma\} \vdash x : \tau$, then $\Gamma, x : \sigma \succ M$.

Proof. Replace in the corresponding type derivation occurrences of (Var) for $x : \tau$ by weakened type derivations of $\{x : \sigma\} \vdash x : \tau$. ☐

As a result of the above Lemma 21, for some type assumptions, such as $(x : \bot)$, it is straightforward to construct from a given term M a term N such that $N \in M|_{(x:\bot)}$ (Lemma 22).

Lemma 22. Let M be a term and let x be a term variable. We have $(\lambda x.M) \in M|_{(x:\bot)}$.

Proof. Let Γ be such that $x \notin \mathrm{dom}(\Gamma)$.

First, assume $\Gamma, x : \bot \succ M$. Therefore, we have $\Gamma \vdash \lambda x.M : \bot \to \tau$ for some type τ, i.e. $\Gamma \succ \lambda x.M$.

Second, assume $\Gamma \succ \lambda x.M$. In the corresponding type derivation the (Abs) rule is applied such that we have $\Gamma, x : \sigma \vdash M : \tau$ for some types σ, τ. By Lemma 21, we obtain $\Gamma, x : \bot \succ M$. ☐

Remark 23. The above proof of Lemma 22 illustrates the main difference between restricted typability and invariant type assumptions [7, Section 5]. In particular, the constructed term $\lambda x.M$ does *not* induce the type assumption $(x : \bot)$ in *every* type derivation, as would be required for an invariant type assumption. In fact, it is challenging to construct a term with $(x : \bot)$ as an invariant type assumption (cf. term J in [7, Lemma 6.3]).

Consider the type assumption $(w : \theta)$ where (for the remainder of this section)

$$\theta := \forall.\, a \to b \to b \to a$$

If we can assign a type σ to a term N and a type τ to a term M, then we can assign the type $\sigma \to \tau$ to the term $w\, M\, N$. This observation is made systematic in the following Definition 24.

Definition 24 ($\langle\cdot\rangle^w_\Delta$)**.** Let w be a term variable and let Δ be an environment. The partial function $\langle\cdot\rangle^w_\Delta : \mathbb{T}_\to \to \mathbb{L}$ is such that

$$\begin{aligned} \langle a\rangle^w_\Delta &= y \quad \text{if } (y : a) \in \Delta \text{ for a unique } y \\ \langle \sigma \to \tau\rangle^w_\Delta &= w\, \langle\tau\rangle^w_\Delta\, \langle\sigma\rangle^w_\Delta \end{aligned}$$

The following Example 25 uses $\langle\cdot\rangle^w_\Delta$ to construct simply typed terms from terms typed by respective simple subtypes.

Example 25. Consider $\Delta := \{y_c : c, y_d : d\}$, $w \in \mathbb{V}$, and $\sigma := ((c \to d) \to c) \to c$.
We have $\langle\sigma\rangle^w_\Delta = w\, y_c\, (w\, y_c\, (w\, y_d\, y_c))$, and

$$\begin{aligned} &\Delta, w : \theta \vdash w\, y_d\, y_c : c \to d \\ &\Delta, w : \theta \vdash w\, y_c\, (w\, y_d\, y_c) : (c \to d) \to c \\ &\Delta, w : \theta \vdash \langle\sigma\rangle^w_\Delta : \sigma \end{aligned}$$

In fact, in the above Example 25 the type σ is the *only* type that can be assigned to $\langle\sigma\rangle^w_\Delta$ in $\Delta, w : \theta$. This is established by the following Lemma 26.

Lemma 26. Let $\sigma \in \mathbb{T}_\to$ be a simple type such that $\mathrm{var}(\sigma) = \{a_1, \ldots, a_n\}$, and let $\Delta := \{y_1 : a_1, \ldots, y_n : a_n\}$.

If $\{y_1 : b_1, \ldots, y_n : b_n, w : \theta\} \vdash \langle\sigma\rangle^w_\Delta : \tau$, then $\tau = \sigma[a_1 := b_1, \ldots, a_n := b_n]$.

Proof. Routine induction on σ. ☐

In order to assume $(w : \theta)$ for typability of a term M, the following Lemma 27 gives a member W of $M|_{(w:\theta)}$. Intuitively, the construction of W relies on three building blocks[5]. First, $J_x[\cdot]$ is used to establish the type assumption $(x : a \to a)$ for some fresh type variable a. Second, the term W_1 is used to establish that for the term variable q the type of $(q\, z)$ is general enough. This excludes type assumptions $(q : \sigma)$ where σ contains too many arrows. Third, the term W_2 is used to establish that the type of $(q\, z)$ is not too general, and can be obtained by specializing θ. This excludes type assumptions $(q : \sigma)$ where σ contains too few arrows or too many distinct type variables. By Lemma 21, this allows for the type assumption $(q : \forall b'. (b' \to b') \to \forall a'. a' \to b' \to b' \to a')$, and the term $K\, (q\, z)\, (\ldots)$ is used to establish the assumption $(w : \theta)$.

Lemma 27. Let M be a term and let w be a term variable. We have $W \in M|_{(w:\theta)}$, where

$$\begin{aligned} W &:= J_x[\lambda z.(\lambda w.M)\, (W_1\, W_2)] \\ &x, z \notin \mathrm{var}(M) \\ W_1 &:= \lambda q.K\, (q\, z)\, (q\, x\, (x\, z)\, (x\, z)\, (x\, z)) \\ W_2 &:= \lambda p\, y_1\, y_2\, y_3.K\, y_1\, (K\, (p\, y_2)\, (p\, y_3)) \end{aligned}$$

Proof. Let Γ be an environment such that $w, x, z \notin \mathrm{dom}(\Gamma)$, and let $c \notin \mathrm{var}(\Gamma)$ (cf. Remark 18).

If $\Gamma, w : \theta \succ M$, then it is easy to verify that $\Gamma, x : c \to c \succ \lambda z.(\lambda w.M)\, (W_1\, W_2)$ using Fact 10 and the type assumptions $p : b' \to b'$, $y_1 : a'$, $y_2 : b'$, $y_3 : b'$, $q : \forall b'. (b' \to b') \to \forall a'. a' \to b' \to b' \to a'$, $z : \bot$, and $w : \theta$ where a', b' are fresh. By Lemma 17, we obtain $\Gamma \succ W$.

For the converse, assume $\Gamma \succ W$. By Lemma 17, we have $\Gamma, x : c \to c \succ \lambda z.(\lambda w.M)\, (W_1\, W_2)$. Therefore, for some σ, τ, $\vec{b}$, σ', τ', and ρ' we have

1) $\Gamma' = \Gamma, x : c \to c, z : \rho'$
2) $\Gamma', q : \sigma \vdash K\, (q\, z)\, (q\, x\, (x\, z)\, (x\, z)\, (x\, z)) : \tau$
3) $\Gamma', q : \sigma \vdash q\, z : \tau$
4) $\sigma = \forall \vec{a}. \sigma' \to \tau'$
5) $\Gamma', p : \sigma' \vdash \lambda y_1\, y_2\, y_3.K\, y_1\, (K\, (p\, y_2)\, (p\, y_3)) : \tau'$
6) $\Gamma', p : \sigma' \vdash \lambda y_1\, y_2\, y_3.y_1 : \tau'$
7) $\Gamma', w : \tau \succ M$

Due to the subterm $(q\, x)$ in (2) we have either $\sigma' = b'$ for some $b' \in \vec{a}$ or $\sigma' = b' \to b''$ for some b', b''. The first case is not possible due to the subterm $(p\, y_2)$ in (5).

In (2) the only derivable type of the subterm $(x\, z)$ is c. Therefore, $\tau' = \forall \vec{b}. a' \to \forall \vec{c}_1. c_1 \to \forall \vec{c}_2. c_2 \to \tau''$ for some type variable sequences $\vec{b}, \vec{c}_1, \vec{c}_2$, some type variables a', c_1, c_2, and some type τ''.

Due to the subterms $(p\, y_2)$ and $(p\, y_3)$ in (5) we have $b' = c_1 = c_2$.

Due to (6) we have $\tau'' = a'$.

Overall, we have $\tau' = \forall \vec{b}. a' \to b' \to b' \to a'$. Therefore, due to (3) we have $\tau = \forall \vec{d}. \rho_1 \to \rho_2 \to \rho_2 \to \rho_1$ for some variable sequence $\vec{d}$ and some types ρ_1 and ρ_2. By Lemma 21 and (7), we obtain $\Gamma, w : \theta \succ M$. □

[5]The construction of W is similar to [7, Lemma 6.4] without the restriction to λI-terms.

Remark 28. Similarly to Remark 23, the type assumption $(w : \theta)$ is not *invariant* across all type derivations for the constructed term W. For example, consider the term $M := w$. It is easy to verify that $\{x : a \to a\} \succ \lambda z.(\lambda w.M)\, (W_1\, W_2)$ using Fact 10 and the type assumptions $p : a \to a$, $y_1 : a$, $y_2 : a$, $y_3 : a$, $q : (a \to a) \to a \to a \to a \to a$, $z : \bot$, and $w : a \to a \to a \to a$. In particular, $(w : \theta)$ is not assumed in a type derivation for the term M. To enforce an invariant type assumption $(w : \theta)$ across all type derivations more machinery would be required.

Already, in the following Lemma 29 we can give a member V of $M|_{(v:\forall.\,\sigma)}$ for any simple type σ. The particular type assumptions $(v : \forall.\, \sigma)$ suffice for a straightforward reduction from semi-unification to typability in Section V. The intuition for the construction of the term $V \in M|_{(v:\forall.\,\sigma)}$ is almost exactly as for the proof of Lemma 27. The only difference being that we now use the term $\langle\sigma\rangle^w_\Delta$ to establish the desired type shape.

Lemma 29 (Prenex Simulation)**.** Let M be a term, let v be a term variable, and let $\sigma \in \mathbb{T}_\to$ be a simple type such that $\{a_1, \ldots, a_n\} = \mathrm{var}(\sigma)$. We have $V \in M|_{(v:\forall.\,\sigma)}$, where

$$\begin{aligned} V &:= J_x[\lambda z.(\lambda v.M)\, (V_1\, V_2)]|_{(w:\theta)} \\ &x, z, w \notin \mathrm{var}(M) \\ V_1 &:= \lambda q.K\, (q\, \underbrace{z \ldots z}_{n \text{ times}})\, (q\, \underbrace{(x\, z) \ldots (x\, z)}_{n \text{ times}}) \\ V_2 &:= \lambda y_1 \ldots y_n.\langle\sigma\rangle^w_{\{y_1:a_1,\ldots,y_n:a_n\}} \end{aligned}$$

Proof. Let Γ be an environment such that $v, x, z, w \notin \mathrm{dom}(\Gamma)$, and let $c \notin \mathrm{var}(\Gamma) \cup \mathrm{var}(\sigma)$ (cf. Remark 18). Wlog. we have $\mathrm{var}(\Gamma) \cap \mathrm{var}(\sigma) = \emptyset$.

If $\Gamma, v : \forall.\, \sigma \succ M$, then it is easy to verify that $\Gamma, x : c \to c, w : \theta \succ \lambda z.(\lambda v.M)\, (V_1\, V_2)$ using Fact 10 and the assumptions $y_1 : a_1, \ldots, y_n : a_n$, $q : \forall.\, a_1 \to \cdots \to a_n \to \sigma$, and $w : \theta$. Therefore, we obtain $\Gamma \succ V$.

For the converse, assume $\Gamma \succ V$. By Lemma 17, we have $\Gamma, x : c \to c, w : \theta \succ \lambda z.(\lambda v.M)\, (V_1\, V_2)$. Therefore, by Lemma 27, for some τ, $\vec{b}_1, \ldots \vec{b}_n$, $\sigma_1, \ldots, \sigma_n$, σ', τ', and ρ' we have

1) $\Gamma' := \Gamma, x : c \to c, w : \theta, z : \rho'$
2) $\Gamma', q : \sigma' \vdash K\, (q\, z \ldots z)\, (q\, (x\, z) \ldots (x\, z)) : \tau$
3) $\Gamma', q : \sigma' \vdash q\, z \ldots z : \tau$
4) $\sigma' = \forall \vec{b}_1. \sigma_1 \to \cdots \to \forall \vec{b}_n. \sigma_n \to \tau'$
5) $\Gamma', y_1 : \sigma_1, \ldots, y_n : \sigma_n \vdash \langle\sigma\rangle^w_{\{y_1:a_1,\ldots y_n:a_n\}} : \tau'$
6) $\Gamma', v : \tau \succ M$

In (2) the only derivable type of the subterm $(x\, z)$ is c. Due to the subterm $(q\, (x\, z) \ldots (x\, z))$ in (2) we have $\sigma_i \in \mathbb{V}$ for $i = 1 \ldots n$. Due to (5), by Lemma 26, we have $\tau' = \sigma[a_1 := \sigma_1, \ldots, a_n := \sigma_n]$.

Due to (3), (4) we have $\tau = \forall \vec{d}. \sigma[a_1 := \rho_1, \ldots, a_n := \rho_n]$ for some variable sequence $\vec{d}$ and some types $\rho_1, \ldots \rho_n$.

By Lemma 21 and (6), we obtain $\Gamma, w : \forall.\, \sigma \succ M$. □

This concludes our necessary System F tool kit.

IV. Semi-unification

Semi-unification is an undecidable [8, Theorem 12] combination of first-order unification and first-order matching. For the undecidability result, it suffices to restrict term syntax to simple types and consider only two term pairs (Problem 30). A pivotal insight of Wells' work is a direct connection between semi-unification and System F type checking [7, Theorem 4.1].

In this section, we recollect necessary definitions and properties of semi-unification from existing work [8], [14]. Additionally, we apply two slight adjustments to semi-unification (which can be found inlined into the monolithic proof of [7, Theorem 4.1]) to further streamline the transition to System F.

The following Problem 30 is the exact restriction of semi-unification used by Wells [7, Theorem 4.1].

Problem 30 (Two-inequality Semi-unification)**.** Given two pairs $(\sigma_1, \tau_1), (\sigma_2, \tau_2) \in \mathbb{T}_{\to}^2$ of simple types, are there substitutions $\varphi, \psi_1, \psi_2 : \mathbb{V} \to \mathbb{T}_{\to}$ with simple codomains such that $\psi_1(\varphi(\sigma_1)) = \varphi(\tau_1)$ and $\psi_2(\varphi(\sigma_2)) = \varphi(\tau_2)$?

Theorem 31 ([8, Remark for Theorem 12])**.** Two-inequality semi-unification (Problem 30) is undecidable.

Remark 32. The known undecidability results for semi-unification rely on either König's lemma [8] or Brouwer's fan theorem [14] (neither of which is provable in axiom-free Coq). However, the argument (reducing Turing machine halting to semi-unification) in the Coq Library of Undecidability Proofs [10] is an improved, axiom-free version of [14].

The following Example 33 illustrates non-structural properties of a solvable semi-unification instance.

Example 33. Consider the semi-unification instance $(a, b \to b), (a \to a, b)$. Intuitively, a solution for the considered instance needs to unify and match a with $b \to b$, and also unify and match $a \to a$ with b, which at first seems structurally contradictory. However, it is solved by substitutions φ, ψ_1, ψ_2 such that $\varphi(a) = a$, $\varphi(b) = a \to a$, $\psi_1(a) = (a \to a) \to a \to a$, and $\psi_2(a) = a$. That is, we have $\psi_1(\varphi(a)) = (a \to a) \to a \to a = \varphi(b \to b)$ and $\psi_2(\varphi(a \to a)) = a \to a = \varphi(b)$.

As our first adjustment, we consider a *left-uniform* restriction (Problem 34) for which the first components of a semi-unification instance are equal.

Problem 34 (Left-uniform, Two-inequality Semi-unification)**.** Given pairs $(\sigma, \tau_1), (\sigma, \tau_2) \in \mathbb{T}_{\to}^2$, are there substitutions $\varphi, \psi_1, \psi_2 : \mathbb{V} \to \mathbb{T}_{\to}$ such that $\psi_1(\varphi(\sigma)) = \varphi(\tau_1)$ and $\psi_2(\varphi(\sigma)) = \varphi(\tau_2)$?

Lemma 35. Two-inequality semi-unification (Problem 30) many-one reduces to left-uniform, two-inequality semi-unification (Problem 34).

Proof. Given pairs $(\sigma_1, \tau_1), (\sigma_2, \tau_2) \in \mathbb{T}_{\to}^2$, let a_1, a_2 be fresh type variables. Construct the simple types $\sigma' := \sigma_1 \to \sigma_2$, $\tau_1' := \tau_1 \to a_2$, and $\tau_2' := a_1 \to \tau_2$. Consider the pairs $(\sigma', \tau_1'), (\sigma', \tau_2')$.

First, assume $\psi_1(\varphi(\sigma_1)) = \varphi(\tau_1)$ and $\psi_2(\varphi(\sigma_2)) = \varphi(\tau_2)$. Construct the substitution φ' such that $\varphi'(a_1) := \psi_2(\varphi(\sigma_1))$, $\varphi'(a_2) := \psi_1(\varphi(\sigma_1))$, and otherwise $\varphi'(a){:=}\varphi(a)$.
We have $\psi_1(\varphi'(\sigma')) = \varphi(\tau_1) \to \varphi'(a_2) = \varphi'(\tau_1')$ and $\psi_2(\varphi'(\sigma')) = \varphi'(a_1) \to \varphi(\tau_2) = \varphi'(\tau_2')$. Therefore, φ', ψ_1, ψ_2 solve the left-uniform instance $(\sigma', \tau_1'), (\sigma', \tau_2')$.

Second, assume $\psi_1(\varphi(\sigma')) = \varphi(\tau_1')$ and $\psi_2(\varphi(\sigma')) = \varphi(\tau_2')$. We have $\psi_1(\varphi(\sigma_1)) = \varphi(\tau_1)$ and $\psi_2(\varphi(\sigma_2)) = \varphi(\tau_2)$. Therefore, φ, ψ_1, ψ_2 solve the given instance $(\sigma_1, \tau_1), (\sigma_2, \tau_2)$. ☐

Corollary 36. Left-uniform, two-inequality semi-unification (Problem 34) is undecidable.

As our second adjustment, we show that the codomain of substitutions φ, ψ_1, ψ_2 can be extended to (not necessarily simple) types. For this, we define the function prune (Definition 37) from types to simple types, which transports solvability of semi-unification (Corollary 39 of the auxiliary Lemma 38).

Definition 37 ($\mathrm{prune}_a : \mathbb{T} \to \mathbb{T}_{\to}$)**.** For $a \in \mathbb{V}$ the function $\mathrm{prune}_a : \mathbb{T} \to \mathbb{T}_{\to}$ is such that

$$\begin{aligned} \mathrm{prune}_a(b) &= b \qquad \text{for } b \in \mathbb{V} \\ \mathrm{prune}_a(\sigma \to \tau) &= \mathrm{prune}_a(\sigma) \to \mathrm{prune}_a(\tau) \\ \mathrm{prune}_a(\forall b.\, \tau) &= a \end{aligned}$$

Lemma 38. Let $\sigma \in \mathbb{T}$, let $\psi : \mathbb{V} \to \mathbb{T}$, and let $a \in \mathbb{V}$ such that $a \notin \mathrm{var}(\sigma)$. There exists a substitution $\psi' : \mathbb{V} \to \mathbb{T}_{\to}$ such that $\psi'(\mathrm{prune}_a(\sigma)) = \mathrm{prune}_a(\psi(\sigma))$.

Proof. For $\psi'(b) := \begin{cases} a & \text{if } b = a \\ \mathrm{prune}_a(\psi(b)) & \text{otherwise} \end{cases}$ the claim follows by routine induction on σ. ☐

The following Corollary 39 shows how for a semi-unification instance a solution in types is transported to a solution in simple types.

Corollary 39. Let $\sigma, \tau \in \mathbb{T}_{\to}$, let $\psi, \varphi : \mathbb{V} \to \mathbb{T}$ such that $\psi(\varphi(\sigma)) = \varphi(\tau)$, and let $a \notin \mathrm{var}(\varphi(\sigma))$. Define the substitution $\varphi' : \mathbb{V} \to \mathbb{T}_{\to}$ such that $\varphi' := \mathrm{prune}_a \circ \varphi$. There exists a substitution $\psi' : \mathbb{V} \to \mathbb{T}_{\to}$ such that

$$\psi'(\varphi'(\sigma)) = \mathrm{prune}_a(\psi(\varphi(\sigma))) = \varphi'(\tau)$$

V. Typability

In this brief section we use prenex simulation (Lemma 29) to reduce left-uniform, two-inequality semi-unification (Problem 34) to System F typability (Problem 6). The construction is analogous to Wells' reduction from semi-unification to System F type checking, and can be understood as partial evaluation of [7, Theorem 4.1] combined with [7, Theorem 6.15].

Lemma 40. Left-uniform, two-inequality semi-unification (Problem 34) many-one reduces to System F typability (Problem 6).

Proof. Given an instance $(\sigma, \tau_1), (\sigma, \tau_2)$ of left-uniform, two-inequality semi-unification, construct the simple types $\sigma' := \sigma \to \sigma$, $\tau_1' := \tau_1 \to \tau_1$, $\tau_2' := \tau_2 \to \tau_2$, and the type environment $\Gamma := \{x : \forall.\, (a \to a) \to a, y : \forall.\, \tau_1' \to \tau_2' \to \sigma'\}$.

Using Lemma 29, construct the terms

$$N \in \big(x\,(\lambda z.y\,z\,z)\big)|_{(x:\forall.\,(a\to a)\to a)}$$
$$M \in N|_{(y:\forall.\,\tau_1'\to\tau_2'\to\sigma')}$$

We show that $(\sigma,\tau_1),(\sigma,\tau_1)$ is solvable iff M is typable.

First, if $(\sigma,\tau_1),(\sigma,\tau_2)$ is solved by φ,ψ_1,ψ_2, then $\Gamma \succ x\,(\lambda z.y\,z\,z)$ using the type assumption $z : \forall.\,\varphi(\sigma')$ and typing y by $\varphi(\tau_1' \to \tau_2' \to \sigma')$. Specifically, $\psi_i(\varphi(\sigma)) = \varphi(\tau_i)$ for $i = 1, 2$ allows the occurrences of z to be typed by $\varphi(\tau_1')$ and $\varphi(\tau_2')$ respectively. Therefore, we obtain $\emptyset \succ M$.

For the converse, assume $\Gamma' \succ M$ for some type environment Γ'. Since $\mathrm{var}(M) = \emptyset$, by Fact 11.4 and Lemma 29 we have that $\Gamma \succ x\,(\lambda z.y\,z\,z)$. Therefore, for some ρ we have $\Gamma, z : \rho \vdash y\,z\,z : \rho$. Therefore, there exists a substitution $\varphi' : \mathbb{V} \to \mathbb{T}$ such that

- $\Gamma, z : \rho \vdash z : \varphi'(\tau_1')$
- $\Gamma, z : \rho \vdash z : \varphi'(\tau_2')$
- $\rho = \forall\vec{a}.\,\varphi'(\sigma')$

By Fact 11.3 there are substitutions $\psi_1', \psi_2' : \mathbb{V} \to \mathbb{T}$ such that $\psi_1'(\varphi'(\sigma')) = \varphi'(\tau_1')$ and $\psi_2'(\varphi'(\sigma')) = \varphi'(\tau_2')$. By Corollary 39 we obtain a solution of $(\sigma,\tau_1),(\sigma,\tau_2)$. □

Corollary 41. System F typability (Problem 6) is undecidable.

VI. Type Checking

This brief section contains a folklore reduction from System F typability (Problem 6) to System F type checking (Problem 7).

Lemma 42. System F typability (Problem 6) many-one reduces to System F type checking (Problem 7).

Proof. Given a term M, let $\{x_1,\dots,x_n\} = \mathrm{var}(M)$, and construct $M' := K\,I\,(\lambda x_1 \dots x_n.M)$, $\Gamma' := \emptyset$, and $\tau' := \forall a.\,a \to a$. We show that M is typable iff $\Gamma' \vdash M' : \tau'$.

First, assume that M is typable, i.e. $\Gamma \vdash M : \tau$ for some type environment Γ and some type τ. By Fact 11.4, we can assume that $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$ for some types $\sigma_1,\dots,\sigma_n$. Applying the (Abs) rule n times we obtain

$$\emptyset \vdash \lambda x_1 \dots x_n.M : \sigma_1 \to \cdots \to \sigma_n \to \tau$$

By Fact 10 we can construct the following type derivation

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash K : \forall a.\,a \to \forall b.\,b \to a}{\Gamma' \vdash K : \tau' \to \forall b.\,b \to \tau'}\,(\text{Inst}) \qquad \Gamma' \vdash I : \tau'}{\Gamma' \vdash K\,I : \forall b.\,b \to \tau'}\,(\text{App})}{\Gamma' \vdash K\,I : (\sigma_1 \to \cdots \to \sigma_n \to \tau) \to \tau'}\,(\text{Inst})$$

Finally, using rule (App) we obtain $\Gamma' \vdash M' : \tau'$.

Second, since M is a subterm of M', a type derivation of $\Gamma' \vdash M' : \tau'$ necessarily contains a type derivation of $\Gamma \vdash M : \tau$ for some type environment Γ and some type τ. □

Corollary 43. System F type checking (Problem 7) is undecidable.

VII. Mechanization

All results in the present work are mechanized using the Coq proof assistant [15] and are integrated into the Coq Library of Undecidability Proofs [10]. Since the library contains a many-one reduction from Turing machine halting to semi-unification (cf. [14]), we obtain many-one reductions from Turing machine halting to both System F typability and System F type checking. Computability and correctness of the corresponding reduction functions is witnessed by the mechanization in axiom-free Coq.

Profiting from the presented simpler undecidability proofs, the overall mechanization encompasses 70 LOC for self-contained problem specification and 1300 LOC for presented arguments. Additionally, it relies on a existing collection (spanning 3000 LOC) of generic System F results containing generation, substitution, subject reduction, and normalization lemmas. Said collection is part of a previously mechanized undecidability result for System F inhabitation [11].

A. Coq Library of Undecidability Proofs

At the core of the Coq Library of Undecidability Proofs is the following mechanized notion of many-one reducibility[6]

```
Definition reduction {X Y} (f : X -> Y)
  (P : X -> Prop) (Q : Y -> Prop) :=
    forall x, P x <-> Q (f x).

Definition reduces {X Y}
  (P : X -> Prop) (Q : Y -> Prop) :=
    exists f : X -> Y, reduction f P Q.

Notation "P ⪯ Q" := (reduces P Q).
```

In the above, a predicate `P` over the domain `X` many-one reduces to a predicate `Q` over the domain `Y`, denoted `P ⪯ Q`, if there exists a function `f : X -> Y` such that for all `x` in the domain `X` we have `P x` iff `Q (f x)`. Implemented in axiom-free Coq any such function `f : X -> Y` is computable. Since Coq's logic is constructive, a proof of `P x <-> Q (f x)` cannot rely on classical principles such as functional extensionality, the law of excluded middle, or choice axioms. As a side note, this approach is compatible with the anti-classical synthetic computability theory [16].

One-tape Turing machine halting is mechanized as `HaltTM 1`[7] as an adaptation [17] of prior work [18] in computability theory, and constitutes the key undecidable problem in the library. Therefore, a mechanized proof of `HaltTM 1 ⪯ Q`, where the predicate `Q` mechanizes a decision problem Q, faithfully witnesses a constructive many-one reduction from Turing machine halting to Q. In fact, the particular many-one reduction function could be extracted from the mechanized proof as a λ-term (in the call-by-value λ-calculus model of computation) using existing techniques [19]. For the formulation of undecidability results, a decision procedure for the the decision problem Q would mechanically induce a decision procedure for the Turing machine halting problem.

[6] `theories/Synthetic/Definitions.v`
[7] `theories/TM/TM.v`

B. Mechanized System F

Curry-style System F type assignment (Definition 4) is mechanized as the inductive predicate[8]

```
type_assignment :
  environment -> pure_term -> poly_type -> Prop
```

where `environment` mechanizes type environments, `pure_term` mechanizes terms, and `poly_type` mechanizes types as follows

```
Inductive pure_term : Type :=
  | pure_var : nat -> pure_term
  | pure_app : pure_term -> pure_term -> pure_term
  | pure_abs : pure_term -> pure_term.

Inductive poly_type : Type :=
  | poly_var : nat -> poly_type
  | poly_arr : poly_type -> poly_type -> poly_type
  | poly_abs : poly_type -> poly_type.

Definition environment := list poly_type.
```

Variable binding is addressed via the unscoped de Bruijn approach [20] supported by the `Autosubst 2` [21] library.

The following predicates `SysF_TYP` and `SysF_TC` mechanize System F typability and type checking respectively.

```
Definition SysF_TYP :
  pure_term -> Prop :=
    fun M => exists Gamma t,
      type_assignment Gamma M t.

Definition SysF_TC :
  environment * pure_term * poly_type -> Prop :=
    fun '(Gamma, M, t) =>
      type_assignment Gamma M t.
```

As a result, Coq proofs of `HaltTM 1` $\preceq$ `SysF_TYP`[9] and `HaltTM 1` $\preceq$ `SysF_TC`[10] witness correctness of the underlying arguments of Corollary 41 and Corollary 43 respectively via rigorous mechanical verification. Constructivity of the corresponding arguments can be mechanically certified using the `Print Assumptions` [22] command.

The main technical contribution, i.e. prenex simulation (Lemma 29), is mechanized as[11]

```
Theorem pure_typable_intro_prenex M s n :
  is_simple s ->
  allfv_poly_type (gt n) s -> { N |
    forall Gamma,
      pure_typable (map tidy
        (many_poly_abs n s :: Gamma)) M <->
      pure_typable (map tidy Gamma) N }.
```

In the above, the arguments are a `pure_term M`, a simple `poly_type s`, and a `nat n` which is an upper bound on the free variables in `s` (mechanized by natural numbers). The result is a computable `pure_term N` such that for any `environment Gamma` typability of `M` in `Gamma` with the additional assumption `many_poly_abs n s` (which abstracts all free variables in `s`) is characterized by typability of `N` in `Gamma`. Notably, the function `tidy` mechanizes the function $[\cdot]_{\forall I}$ from Lemma 13 restricting typability to the $\forall$I-fragment of System F.

[8]`theories/SystemF/SysF.v`

[9]`theories/SystemF/Reductions/HaltTM_1_to_SysF_TYP.v`

[10]`theories/SystemF/Reductions/HaltTM_1_to_SysF_TC.v`

[11]`theories/SystemF/Util/pure_typable_prenex.v`

C. Challenges and Design Decisions

Conceptualizing and mechanizing a proof, there are at least two different goals that can be of interest.

On the one hand, it is valuable to have a concise mechanization for the exact problem at hand using the simplest argument. From a software engineering point of view, such a focused approach is easier to survey, maintain, adapt, and integrate.

On the other hand, it is worthwhile to develop a universal framework, containing broader concepts. Such concepts may improve the overall understanding of the problem at hand and can be used for similar problems. Such a general approach puts more emphasis on the methods (rather than merely verifying a single result), and is likely to contribute to new insights.

While present work aims for the former goal, the traditional argument by Wells [7] is along the lines of the latter. Initially, the author tried to mechanize the exact traditional argument using the Coq proof assistant. However, such a mechanization has to overcome several technical difficulties.

First, the pervasive use of contexts (terms with a variable-capturing holes) hinders a mechanization based upon the de Bruijn or locally nameless term representations. Additionally, global type environments used by invariant type assumptions [7, Definition 5.1] depend on the actual names of bound variables. Therefore, it is difficult to utilize existing, mature infrastructure (e.g. `Autosubst 2`) for variable binding[12].

Second, forming canonical type expressions [7, Definition 3.10] requires non-local (wrt. type syntax, assuming monadic binding) operations such as sorting of abstracted variables[13]. In the structurally recursive setting of Coq, it requires extra infrastructure to argue modulo an equivalence relation with non-substructural normalization.

Third, for the construction of invariant type assumptions, it is relevant which intermediate judgments are part of a type derivation (cf. [7, Definition 5.1]). Therefore, type assignment cannot be treated entirely at the level of (proof-irrelevant) propositions.

While none of the above technical difficulties are prohibitive, their solution requires additional effort before any argument regarding invariant type assumptions is mechanized. In sum, it boils down to the question[14]:

> *What is a suitable presentation of terms, types, and type assignment in order to mechanize invariant type assumptions?*

The present work came into existence because the author failed to answer this question. In particular, the key design decision for the the presented mechanized undecidability results was to rely on the off-the-shelf de Bruijn representation, as provided by the `Autosubst 2` library. This forced a fundamental revision of the concept of invariant type assumptions, leading to its weaker restricted typability alternative. Most importantly, for

[12]For a discussion on existing approaches see [23].

[13]The normalization procedure [7, Definition 3.10] is not compatible with the de Bruijn type representation, because the order of abstractions is dictated by the representation.

[14]This question resembles the POPLmark challenges [24].

restricted typability names of bound variables are immaterial. Further, it relies only on top-level judgments, and therefore is easily presented as a (proof-irrelevant) proposition. Still, this restricted setting suffices to inspect the decision problems at hand, and ultimately, leads to simpler proofs. As added benefit, the mechanization does not require additional ingenuity, since most proofs are carried out simply by structural induction and repeated application of inversion principles. Further, it is easy to see that the presented predicates `type_assignment`, `SysF_TYP`, and `SysF_TC` faithfully mechanize System F type assignment, typability, and type checking.

VIII. Conclusion

Compared to Wells' monumental technical argument for the undecidability (and equivalence) of System F typability and type checking, the presented proofs seem rather mundane. The contributing observation is that a direct reduction from (a fragment of) semi-unification to System F typability (without System F type checking as an intermediate step) can be achieved by comparatively simple means. Admittedly, the new approach is a one-trick pony that is hardly suited for any other result, while Wells' calculus of invariant type assumptions is of more general interest. Also, in favor of a simpler mechanization, the new approach does not adopt canonical type expressions, which are essential for the development of the original argument. The presented prenex simulation lemma (the main technical contribution of the present work) is a much weaker version of an existing result [7, Theorem 6.14]. In particular, it cannot handle free type variables or nested quantification, and it cannot be used to control the shape of type assumptions across all type derivations.

Still, the presented weaker approach suffices to show the undecidability of System F typability and type checking, with comparatively little effort. Therefore, it is easier to survey [9], teach, and mechanically verify. Notably, the provided mechanization witnesses correctness and constructivity (in the sense of axiom-free Coq) of the overall argument. For free (relying on previously mechanized results), integration into the Coq library of undecidability proofs establishes mechanically verified, constructive many-one reductions from Turing machine halting to both typability and type checking, implying their many-one equivalence. Yet, the task to mechanize Wells' broader theory of invariant type assumptions for a comprehensive verification of properties of System F remains open.

The restricted nature of properties used, raises an interesting question: is there a natural, minimal, conservative fragment of System F exposing undecidable typability (and type checking)?

Acknowledgment

The author is grateful for the insightful discussions with Jakob Rehof, Paweł Urzyczyn, and Joe Wells on decision problems for System F and their history. In addition, author appreciates the support of Yannick Forster with both the `Autosubst 2` tool and the Coq Library of Undecidability Proofs.

References

[1] J. Girard, "Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur," Ph.D. dissertation, Université Paris VII, 1972.

[2] J. C. Reynolds, "Towards a theory of type structure," in *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, ser. Lecture Notes in Computer Science, B. Robinet, Ed., vol. 19. Springer, 1974, pp. 408–423. [Online]. Available: https://doi.org/10.1007/3-540-06859-7_148

[3] H. Barendregt, "Introduction to generalized type systems," *J. Funct. Program.*, vol. 1, no. 2, pp. 125–154, 1991.

[4] M. H. Sørensen and P. Urzyczyn, *Lectures on the Curry-Howard Isomorphism*, ser. Studies in Logic and the Foundations of Mathematics. Elsevier, 2006, vol. 149.

[5] D. Leivant, "Polymorphic type inference," in *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*, J. R. Wright, L. Landweber, A. J. Demers, and T. Teitelbaum, Eds. ACM Press, 1983, pp. 88–98. [Online]. Available: https://doi.org/10.1145/567067.567077

[6] J. B. Wells, "Typability and Type-Checking in the Second-Order lambda-Calculus are Equivalent and Undecidable," in *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*. IEEE Computer Society, 1994, pp. 176–185. [Online]. Available: https://doi.org/10.1109/LICS.1994.316068

[7] ——, "Typability and Type Checking in System F are Equivalent and Undecidable," *Ann. Pure Appl. Log.*, vol. 98, no. 1-3, pp. 111–156, 1999. [Online]. Available: https://doi.org/10.1016/S0168-0072(98)00047-5

[8] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn, "The undecidability of the semi-unification problem," *Inf. Comput.*, vol. 102, no. 1, pp. 83–101, 1993. [Online]. Available: https://doi.org/10.1006/inco.1993.1003

[9] O. B. Bassler, "The surveyability of mathematical proof: A historical perspective," *Synth.*, vol. 148, no. 1, pp. 99–133, 2006. [Online]. Available: https://doi.org/10.1007/s11229-004-6221-7

[10] Y. Forster, D. Larchey-Wendling, A. Dudenhefner, E. Heiter, D. Kirst, F. Kunze, G. Smolka, S. Spies, D. Wehr, and M. Wuttke, "A Coq Library of Undecidable Problems," in *The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020)*, 2020. [Online]. Available: https://github.com/uds-psl/coq-library-undecidability

[11] A. Dudenhefner and J. Rehof, "A simpler undecidability proof for system F inhabitation," in *24th International Conference on Types for Proofs and Programs, TYPES 2018, June 18-21, 2018, Braga, Portugal*, ser. LIPIcs, P. Dybjer, J. E. Santo, and L. Pinto, Eds., vol. 130. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 2:1–2:11. [Online]. Available: https://doi.org/10.4230/LIPIcs.TYPES.2018.2

[12] P. Giannini and S. Ronchi Della Rocca, "Characterization of typings in polymorphic type discipline," in *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, 1988, pp. 61–70. [Online]. Available: https://doi.org/10.1109/LICS.1988.5101

[13] R. Milner, "A theory of type polymorphism in programming," *J. Comput. Syst. Sci.*, vol. 17, no. 3, pp. 348–375, 1978. [Online]. Available: https://doi.org/10.1016/0022-0000(78)90014-4

[14] A. Dudenhefner, "Undecidability of Semi-Unification on a Napkin," in *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, ser. LIPIcs, Z. M. Ariola, Ed., vol. 167. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 9:1–9:16. [Online]. Available: https://doi.org/10.4230/LIPIcs.FSCD.2020.9

[15] T. C. D. Team, "The Coq proof assistant, version 8.12.0," Jul. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4021912

[16] A. Bauer, "First steps in synthetic computability theory," in *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 2005, Birmingham, UK, May 18-21, 2005*, ser. Electronic Notes in Theoretical Computer Science, M. H. Escardó, A. Jung, and M. W. Mislove, Eds., vol. 155. Elsevier, 2005, pp. 5–31. [Online]. Available: https://doi.org/10.1016/j.entcs.2005.11.049

[17] Y. Forster, F. Kunze, and M. Wuttke, "Verified programming of turing machines in Coq," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, J. Blanchette and C. Hritcu, Eds. ACM, 2020, pp. 114–128. [Online]. Available: https://doi.org/10.1145/3372885.3373816

[18] A. Asperti and W. Ricciotti, "A formalization of multi-tape Turing machines," *Theor. Comput. Sci.*, vol. 603, pp. 23–42, 2015. [Online]. Available: https://doi.org/10.1016/j.tcs.2015.07.013

[19] Y. Forster and F. 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, September 9-12, 2019, Portland, OR, USA*, ser. LIPIcs, J. Harrison, J. O'Leary, and A. Tolmach, Eds., vol. 141. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 17:1–17:19. [Online]. Available: https://doi.org/10.4230/LIPIcs.ITP.2019.17

[20] N. G. De Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem," in *Indagationes Mathematicae (Proceedings)*, vol. 75, no. 5. North-Holland, 1972, pp. 381–392.

[21] K. Stark, S. Schäfer, and J. Kaiser, "Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions," in *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, A. Mahboubi and M. O. Myreen, Eds. ACM, 2019, pp. 166–180. [Online]. Available: https://doi.org/10.1145/3293880.3294101

[22] "The Coq Proof Assistant Reference Manual," https://coq.inria.fr/distrib/current/refman/, accessed: 2020-07-30.

[23] K. Stark, "Mechanising syntax with binders in Coq," Ph.D. dissertation, Saarland University, Jan 2020.

[24] A. Abel, G. Allais, A. Hameer, B. Pientka, A. Momigliano, S. Schäfer, and K. Stark, "Poplmark reloaded: Mechanizing proofs by logical relations," *Journal of Functional Programming*, vol. 29, p. e19, 2019.