Automation of Higher-Order Logic

Authors: Christoph Benzmüller and Dale Miller

Readers: Peter Andrews, Jasmin Blanchette, William Farmer,
Herman Geuvers, and Bruno Woltzenlogel Paleo

Venue: The Handbook of the History of Logic, eds. D. Gabbay & J. Woods
Volume 9: Logic and Computation, editor Jörg Siekmann

Contents

1. Introduction

Early efforts to formalize mathematics in order to make parts of it more rigorous and to show its consistency started with the codification of parts of logic. There was work on the logical connectives by, for example, Boole and Pierce, and later work to formalize additionally quantifiers (Frege, Church, Gentzen, etc.). Once the basic concepts of logic—logical connectives, (first-order) quantification, and inference rules—were formalized, the consistency of various first-order logics was established by Gödel's completeness theorem (1930a) and Gentzen's cut-elimination theorem (1969a; 1969b). Equipped with such logical systems, logicians turned to the formalizations of mathematics that had been started by Peano (1889) and Hilbert (1899) and attempted to encode the objects of mathematics, such as real numbers, sets, groups, etc., by building them on top of logic.

There are several ways to undertake such formalizations. One early and successful approach involved building various theories for, say, Zermelo-Fraenkel set theory, as first-order logic extended with axioms postulating the existence of certain sets from the existence of other sets. Instead of sets, one could also explore the use of algebra and universal properties to develop a categorical theory of mathematics. This chapter addresses yet another approach to formalizing mathematical concepts on top of quantification logic: one can use higher-order quantification instead of only first-order quantification. In the syntax of first-order logic, there are terms and predicates: the terms denote individuals of some intended domain of discourse, and predicates denote some subset of that domain. Inspired by set theory, it is also natural to ask if certain predicates hold of other predicates, e.g., is a given binary relation transitive or not. Other natural notions, such as Leibniz's equality—which states that x is equal to y if

every predicate true of x is also true of y—would naturally be written as the formula $\forall P. Px \supset Py$.[1]

Such higher-order quantification was developed first by Frege and then by Russell in his ramified theory of types, which was later simplified by others, including Chwistek and Ramsey, Carnap, and finally Church in his simple theory of types (STT), also referred to as classical higher-order logic.

1.1. Formalizing set comprehension as λ-abstraction

Church's STT (Church, 1940), which is the focus of this chapter, based both terms and formulas on simply typed λ-terms and the equality of terms and formulas is given by equality of such λ-terms. The use of the λ-calculus had at least two major advantages. First, λ-abstractions over formulas allow the explicit naming of sets and predicates, something that is achieved in set theory via the comprehension axioms. For example, if $\varphi(x)$ is a formula with one free variable x, then the set comprehension axiom provides the existence of the set $\{x \in A \mid \varphi(x)\}$, for some set A. Typed λ-abstraction achieves this in a simple step by providing the term $\lambda x.\varphi(x)$: here, the variable x is given a type that, in principle, can be identified with the same set A. Second, the complex rules for quantifier instantiation at higher-types is completely explained via the rules of λ-conversion (the so-called rules of α-, β-, and η-conversion) which were proposed earlier by Church (1932,1936).

Higher-order substitution can be seen (from the inference step point-of-view) as one step, but it can pack a significant computational influence on a formula: normalization of λ-terms can be explosive (and expressive), in particular, since λ-terms may contain logical connectives and quantifiers. Bindings are also treated uniformly for all structures and terms that have bindings. For example, if p is a variable of predicate type and A is the formula $\forall p. B(p)$, then the universal instantiation of A with the term, say, t, namely the formula $[t/p]B$, can be a formula with many more occurrences of logical connectives and quantifiers than there are in the formula $B(p)$.

1.2. Packing more into inference rules

Given that fewer axioms are needed in STT than in axiomatic set theory, and that the term and formula structure is enriched, some earlier researchers in automated theorem proving were attracted to STT since traditionally such early provers were not well suited to deal with large numbers of axioms. If small theories could be achieved by making the notion of terms and formula more complex, this seemed like a natural choice. Thus, if one extended, say, first-order resolution with a more complex form of unification (this time, on λ-terms), then one might be addressing theorems that would otherwise need explicit notions of sets and their axioms.

[1] Occasionally we use the . notation in this paper to separate the body of quantified formulas or λ-abstractions from the binder, and parentheses may be avoided if the formulas structure is obvious. An alternative notation for $\forall P. Px \supset Py$ would thus be $\forall P(Px \supset Py)$.

Gödel (1936) pointed out that higher-order logic (actually, higher-order arithmetic) can yield much shorter proofs than are possible in first-order logic. Parikh (1973) proved this result years later: in particular, he proved that there exist arithmetical formulas that are provable in first-order arithmetic, but whose shortest proofs in second-order arithmetic are arbitrarily smaller than any proof in first-order. Similarly, Boolos (1987) presented a theorem of first-order logic comprising about 60 characters but whose shortest proof (allowing the introduction and use of lemmas) is so large that the known size of the universe could not be used to record it: on the other hand, a proof of a few pages is possible using higher-order logic.

The embedding of λ-conversion within inference rules that is available within STT is related to modern approaches to making inference rules more expressive by placing computation into them. A modern updating of this approach is found in the work on *deduction modulo* (Dowek et al., 2003; Cousineau and Dowek, 2007) where functional programming style computations on formulas and terms are permitted within inference steps. Indeed, the connection between Church's approach to using higher-order quantification and λ-terms can be closely simulated using deduction modulo (Burel, 2011a). Recent developments in *focused proof systems* (Andreoli, 1992; Liang and Miller, 2009) provides a means of defining large scale inference rules that include possibly non-deterministic computation (Miller, 2011).

1.3. Plan of this chapter

We refer the reader looking for more details about higher-order logic and STT to the textbook of Andrews (2002) and to the handbook and encyclopedia articles by Andrews (2001), Andrews (2009), Enderton (2012), and Leivant (1994). Another recommended article has been contributed by Farmer (2008). Here we shall focus on the issues surrounding theorem proving, particularly, automated theorem proving in subsets and variants of Church's STT. In particular, in Section 2, we describe some of the history and the background of formal treatments of quantification and the closely associated notions of binding and substitution. In Section 3, we present the technical description of STT. The meta-theory of STT, including general models and cut-elimination, are addressed in Section 4. Skolemization, unification, pre-unification, and pattern unification, which are central concepts for proof automation, are discussed in Section 5. Section 6 then addresses core challenges for higher-order automated theorem provers, such as substitutions for predicate variables and the automation of extensionality. An overview of interactive and automatic theorem provers of (fragments of) STT is presented in Section 7. We conclude briefly in Section 8.

2. Formalization of quantificational logic

Quantification is a key feature of natural language and its treatment has been widely studied by linguists and logicians. A core interest has been to appropriately match informal use of quantificational expressions in natural languages

with their formal treatment in logic formalisms. In that context, quantification also plays a pivotal role. However, the focus is on the widely adopted traditional notion of universal and existential quantification only.[2] A crucial question is what kind of objects an existential or universal quantifier may range over, or, in other words, what kind of objects the universe may contain. In classical first-order logic these objects are strictly of elementary nature, like the person 'Bob' or the number '5'; we call them first-order objects. Not allowed are quantifications over properties of objects (these are identified with sets of objects) or functions. In higher-order logic, quantification is not restricted to only elementary objects: quantification over sets or functions of objects is generally allowed. Peano's induction principle for the natural numbers is a good example. Using quantification over first-order objects ($\forall x$) and over properties (i.e., sets) of first-order objects ($\forall P$), this principle can be elegantly expressed in higher-order logic by the axiom

$$\forall P.\, P0 \supset (\forall x.\, (Px \supset P(s\ x)) \supset \forall y.\, Py),$$

where s denotes the successor function. This formula belongs to second-order logic, since the variable P ranges only over sets of first-order objects. In higher-order logic one may move arbitrarily up in this hierarchy; that is, quantifications over sets of sets of objects, *etc*, are allowed. First-order and second-order logic are, in this sense, fragments of higher-order logic.

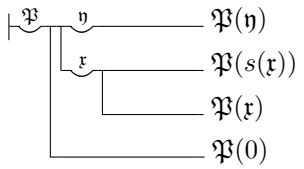
There are significant theoretical and practical differences between first-order logic and higher-order logic regarding, for example, expressive power and proof theory. These fundamental differences—some of which will be outlined in the next sections—have alienated many logicians, mathematicians, and linguists. A (rather unfortunate) consequence has been that the community largely focused on the theory and mechanization of first-order logic. In particular automation of higher-order logic, the topic of this article, has often been considered as too challenging, at least until recently.

2.1. Earliest work on higher-order logic

Publication of Frege's (1879) *Begriffsschrift* is commonly considered the birth of modern symbolic logic. The *Begriffsschrift* presents a 2-dimensional formal notation for predicate calculus and develops an adequate and still relevant notion of formal proof. Frege's notation for universal quantification appropriately marks both the bound variable and the scope of quantification. Most importantly, quantification in Frege's notation is not restricted to first-order

[2] Traditional quantification and generalized quantification are contrasted in the text by Westerståhl (2011); see also the references therein.

objects, and in his notation the above induction axiom can be formalized as:[3]



Thus, the *quantification over predicate, relation and function symbols* is explicitly allowed in the language of the *Begriffsschrift*. A representative example of such quantification is Frege's statement (76) in the *Begriffsschrift*, which described the transitive closure of a relation.

The fact that Frege's logic of the *Begriffsschrift* is indeed higher-order can also be retraced by following some of his substitutions for relation symbols. For example, in his derivation of statement (70), Frege substitutes a relation symbol f with a function of one argument. In modern notation his concrete instantiation can be expressed by the lambda term $\lambda z.\, Fz \supset \forall a.\, fza \supset Fa$. The support for such kind of *higher-order substitutions* is another distinctive feature of higher-order logic. Frege carries out this substitution and he implicitly applies normalization. He does not give, however, a sufficiently precise definition of how such substitutions are applied.

It was Bertrand Russell (1902; 1903) who first pointed out that unrestricted quantification, as considered by Frege, in connection with the comprehension principles,[4] enables the encoding of paradoxes and leads to inconsistency. The most prominent such paradox, widely known as *Russell's paradox*, involves the set of all non-self-containing sets. Russell (1908) suggested a few years later a *ramified theory of types* as a paradox-free basis for the formalization of mathematics that differentiates between objects and sets (or functions) consisting of these kinds of objects. On one hand, Russell was trying to avoid the paradoxes that had plagued earlier work and he attributed some of the paradoxes to a *vicious circle principle* in which some mathematical objects are defined in terms of collections that include the object being defined. In modern terms, Russell wanted to disallow *impredicative definitions* and ramified types were one way to avoid such impredicativity. On the other hand, Russell was trying to reduce mathematics to logic and since ramification made it difficult to encode mathematics, Russell introduced the *axiom of reducibility* which essentially collapses ramifications and allows one to still make impredicative definitions (Ramsey, 1926; Chwistek, 1948).

The ramified theory of types was subsequently selected by Russell & White-

---

[3]Here $\overset{\mathfrak{y}}{\smile}\!\!\!-\ \mathfrak{P}(\mathfrak{y})$ corresponds to $\forall y.\, Py$ and $\begin{array}{l}\text{┌─}\ \mathfrak{P}(s(\mathfrak{x}))\\ \text{└─}\ \mathfrak{P}(\mathfrak{x})\end{array}$ to $Px \supset Psx$; the rest is obvious. The vertical bar on the left marks that the entire statement is asserted.

[4]The comprehension principles assure the existence of abstractions over formula expressions; an example for a type restricted comprehension axiom (schema) is $\exists u_{\alpha^1 \to \ldots \to \alpha^n \to o} . \forall x^1 \ldots \forall x^n . (ux^1 \ldots x^n) = b_o$.

head as the logical foundation for the *Principia Mathematica* (Whitehead and Russell (1910, 1912, 1913)). They shared the philosophical standpoint of *logicism*, initiated by the work of Frege, and they wanted to show that all mathematics can be reduced to logic. The *Principia* succeeded to a significant extent and an impressive number of theorems of set-theory and finite and infinite arithmetic, for example, were systematically derived from the *Principia*'s logical foundations. However, the *Principia* was also criticized for its use of the axiom of reducibility and its use of the *axiom of infinity*, which asserts that a set with infinitely many objects exists. It thus remained debatable what *Principia* actually demonstrated: was it a reduction of mathematics to logic or a reduction of mathematics to some (controversial) set theory?

In the 1920s, a number of people suggested a *simple theory of types* as an alternative to Russell's ramified type theory.[5] These suggestions led to the seminal paper by Church (1940), which will be addressed in some detail in the next section. The terms *simple theory of types* and *classical higher-order logic* typically refer to the logic presented in Church (1940).

It should be remarked that the idea of employing a type hierarchy can, to some extent, be attributed to Frege: in his writings he usually mentions explicitly the kind of objects—predicates, predicates of predicates, etc.—a quantified variable is representing (cf. Quine (1940)).

In summary, higher-order logic is an expressive formalism that supports quantification over predicate, relation, and function variables and that supports complex substitutions of such variables. Such a rich language has several pitfalls with which to be concerned. One such pitfall involves providing a technically precise and sound notion of substitution involving bindings. Another (more important) pitfall involves the careful treatment of self-referential, impredicative definitions since these may lead to inconsistencies. A possible solution to the latter pitfall is to consider syntactical restrictions based on type hierarchies and to use these to rule out problematic impredicative definitions.

2.2. Different notions of higher-order logic

The notion of *higher-order* when applied to logic formalisms is generally not as unambiguous as the above text might suggest. We mention below three different groups of people who appear to use this term in three different ways.

Philosophers of mathematics often distinguish between first-order logic and second-order logic only. The latter logic, which is used as a formal basis for all of mathematics, involves quantification over the domain of all possible functions. In particular Kurt Gödel's work draws an important theoretical line between first- and second-order logic. Shortly after proving completeness of first-order logic (1929; 1930b), Gödel presented his celebrated first incompleteness theorem (1931). From this theorem it follows that second-order logic is necessarily

[5] In Church (1940), Church attributes the simple theory of types to Chwistek, Ramsey, and, ultimately, Carnap. Simple type theory corresponds to the ramified theory of type plus the axiom of reducibility.

incomplete: that is, truth in higher-order logic can not be recursively axiomatized. Thus, higher-order logic interpreted in this sense consists largely of a model-theoretic study, typically of the *standard model of arithmetic* (cf. Shapiro (1985)).

Proof-theoreticians take logic to be synonymous with a formal system that provides a recursive enumeration of the notion of theoremhood. A higher-order logic is understood no differently. The distinctive characteristic of such a logic, instead, is the presence of predicate quantification and of comprehension (i.e., the ability to form abstractions over formula expressions). These features, especially the ability to quantify over predicates, profoundly influence the proof-theoretic structure of the logic. One important consequence is that the simpler induction arguments of cut-elimination that are used for first-order logic do not carry over to the higher-order setting and more sophisticated techniques, such as the "candidats de réductibilité" due to Jean-Yves Girard (1971), must be used. Semantical methods can also be employed, but the collection of models must now include *non-standard models* that use restricted function spaces in addition to the standard models used for second-order logic.

Implementers of deduction systems usually interpret higher-order logic as any computational logic that employs λ-terms and quantification at higher-order types, although not necessarily at predicate types. Notice that if quantification is extended only to non-predicate function variables,[6] then the logic is similar to a first-order one in that the cut-elimination process can be defined using an induction involving the sizes of (cut) formulas. However, such a logic may incorporate a notion of equality based on the rules of λ-conversion and the implementation of theorem proving in it must use (some form of) higher-order unification.

3. Church's simple theory of types (classical higher-order logic)

3.1. The λ-calculus as computation (middle and late 1930s)

The λ-calculus is usually thought of as a framework for computing functions. In the setting of STT, however, where the discipline of *simple types* is applied, those functions are greatly limited in what they can compute. A typical use of λ-conversion in STT is to provide the function over syntax that instantiate a quantified formula with a term. If one wants to describe the function that, for example, returns the smallest prime divisor of an integer, one would specify relational specifications of primality, division, etc., and then show that such relations are, in fact, total functions. Thus, the foundations that STT provides to mathematics is based on *relations*: this is in contrast to, say, the *function*-centric foundation of Martin-Löf type theory (Martin-Löf, 1982). It is worth pointing out that although typed λ-calculi are considered the quintessential proof structure for intuitionistic logic, Church, as the father of the λ-calculus, has shown

[6] This is meant to also exclude, for example, $\forall F_{(\iota\to o)\to\iota}$, where predicates can be used as arguments.

little interest in intuitionistic logic himself: in particular, his development of STT was based on classical logic.

3.2. Mixing λ-calculus and logic

Church applied his λ-calculus to the description of not only quantificational structures and higher-order substitution but also many familiar mathematical constructions. For example, the usual notion for membership $x \in P$, i.e., "x is a member of the set P", can be written instead using the notion Px which is familiar from first-order logic, i.e., "the predicate P is true of x". Thus, the concept of set is represented not as a primitive concept itself but is constructed using logic. Of course, to allow interesting constructions involving sets, we need a higher-order logic that allows operations on predicates that can mimic similar operations on sets. For example, if predicates A and B denote sets then the expressions $\lambda x. Ax \wedge Bx$ and $\lambda x. Ax \vee Bx$ denote the intersection and union of the sets described by these predicates. Furthermore, $\lambda C. \forall x. Cx \supset Ax$ describes the power-set of A (i.e., the set of sets C that are subsets of A). We can even use λ-abstractions to make more abstractions possible: the notion of set union, for example, can be defined as the λ-abstraction $\lambda A. \lambda B. \lambda x. Ax \wedge Bx$ and the notion of power-set can be $\lambda A. \lambda C. \forall x. Cx \supset Ax$.

As Kleene and Rosser (1935) discovered, a direct mixing of the λ-calculus with logic can lead to an inconsistent logic. One of the simplest presentations of an inconsistency arising from mixing the untyped λ-calculus with (classical) logic is called Curry's paradox (Curry, 1942). Let y be a formula and let r be the λ-abstraction $\lambda x.xx \supset y$. Via λ-conversion rr is equal and, hence, equivalent to $rr \supset y$. Hence, we have the two implications $rr \supset (rr \supset y)$ and $(rr \supset y) \supset rr$. From the former we get (by contracting assumptions) $rr \supset y$, and hence, by modus ponens with the latter we know rr. By a further modus ponens step we thus get y. Since y was an arbitrary formula, we have proved a contradiction.

One way to avoid inconsistencies in a logic extended with the λ-calculus is to adopt a variation of Russell's use of types (thereby eliminating the self application rr in the above counterexample). When Church modified Russell's ramified theory of types to a "simple theory" of types, the core logic of this chapter was born (Church, 1940). Mixing λ-terms and logic as is done in STT permits capturing many aspects of set theory without direct reference to axioms of set theory.

There are costs to using the strict hierarchy of sets enforced by typing: no set can contain both a member of A as well as a subset of A. The definition of subset is based on a given type: asking if a set of integers is a subset of another set of integers is necessarily different than asking if a binary relation on integers is a subset of another set of binary relations on integers.

3.3. Simple types and typed λ-terms

The *primitive types* are of two kinds: o is the type of propositions and the rest are the types of basic domains of individuals: thus we are adopting a many-sorted approach to logic. However, analogously to Church (1940), we shall just

admit one additional primitive type, namely, ι, for the domain of individuals which correspond to the domain of first-order variables. (Admitting more than this one additional primitive type is no challenge.) The set of *type expressions* is the least set containing the primitive types and such that $(\gamma \to \tau)$ is a type expression whenever γ and τ are type expressions. Here, types of the form $(\gamma \to \tau)$ denote the type of functions with domain type γ and codomain type τ. If parentheses are omitted, we shall assume that the arrow constructor associates to the right: i.e., $\delta \to \gamma \to \tau$ denotes $(\delta \to (\gamma \to \tau))$. The order $ord(\tau)$ of a type τ is defined by structural recursion:[7] if τ is a primitive type, then $ord(\tau) = 0$; otherwise $ord(\gamma \to \tau) = \max(ord(\gamma) + 1, ord(\tau))$. Thus, the order of $\iota \to \iota \to \iota$ is 1 and the order of $(\iota \to \iota) \to \iota$ is 2.

Let Σ be a set of typed constants, i.e., tokens with a subscript that is a simple type and denote by Σ_τ the subset of Σ of constants with subscript τ. Lowercase letters with subscripts, e.g., c_τ, are syntactic variables ranging over constants with subscript τ. For each type τ, let $\mathcal{V}_\tau$ be an infinite set of variables $x^1_\tau, x^2_\tau, \ldots$, all with subscript τ. The uppercase letters X, Y, and Z with type expression subscripts are syntactic variables ranging over particular variables: e.g., X_τ is a syntactic variable ranging over the particular variables in $\mathcal{V}_\tau$. Subscripts of syntactic variables may be omitted when they are obvious or irrelevant in a particular context. Given the constants in Σ and variables in $\mathcal{V}_\tau$ ($\tau \in \mathcal{T}$), we can now define the binary relation of a term with a type as the least relation satisfying the following clauses.

1. If $c_\tau \in \Sigma$ then c_τ is a term of type τ.
2. If $X_\tau \in \mathcal{V}_\tau$ then X_τ is a term of type τ.
3. If X_τ is a variable with subscript τ and M_γ is a term of type γ, then $(\lambda X_\tau\, M_\gamma)$ is a term of type $(\tau \to \gamma)$.
4. If $F_{\tau\to\gamma}$ is a term of type $(\tau \to \gamma)$, and A_τ is a term of type τ, then $(F_{\tau\to\gamma}\, A_\tau)$ is a term of type γ.

The uppercase letters A, B, C, and M with type expression subscripts, e.g., M_τ, are syntactic variables ranging over terms of their displayed type. The parentheses in $(\lambda X_\tau\, M_\gamma)$ and $(F_{\tau\to\gamma}\, A_\tau)$ can be omitted if it is clear from context how to uniquely insert them.

Each occurrence of a variable in a term is either *bound* by a λ or *free*. We consider two terms A and B to be equal (and write $A \equiv B$), if they are the same up to a systematic change of bound variable names (i.e., we consider α-conversion implicitly). A term A is closed if it has no free variables.

A substitution of a term A_α for a variable X_α in a term B_β is denoted by $[A/X]B$. Since we consider α-conversion implicitly, we assume that the bound variables of B are appropriately renamed to avoid variable capture. For example, $[x^1_\iota / x^2_\iota]\lambda x^1 . x^2$ is equal to, say, $\lambda x^3 . x^1$, which is not equal to $\lambda x^1 . x^1$.

[7] Different notions of 'order' have actually been discussed in the literature. We may, e.g., start with $ord(\iota) = 0$ and $ord(o) = 1$.

Two important relations on terms are given by β-reduction and η-reduction. A β-redex $(\lambda X A)B$ (i.e., the application of an abstraction to an argument) β-reduces to $[B/X]A$ (i.e., the substitution of an actual argument for a formal argument). If X is not free in C, then $\lambda X(CX)$ is an η-redex and it η-reduces to C. If A β-reduces to B then we say that B β-expands to A. Similarly, if A η-reduces to B then we say that B η-expands to A. For terms A and B of the same type, we write $A{\equiv_\beta}B$ to mean A can be converted to B by a series of β-reductions and expansions. Similarly, $A{\equiv_{\beta\eta}}B$ means A can be converted to B using both β- and η-conversion. For each simply typed λ-term A there is a unique β-normal form (denoted $A{\downarrow_\beta}$) and a unique $\beta\eta$-normal form (denoted $A{\downarrow_{\beta\eta}}$). From this fact we know $A \equiv_\beta B$ ($A \equiv_{\beta\eta} B$) if and only if $A{\downarrow_\beta} \equiv B{\downarrow_\beta}$ ($A{\downarrow_{\beta\eta}} \equiv B{\downarrow_{\beta\eta}}$).

The simply typed λ-terms of Church (1940) are essentially the ones in common use today (cf. Barendregt (1997); Barendregt et al. (2013)). One subtlety is that all variables and constants carry with them their type as part of their name: that is, constants and variable are not associated with a type which could vary with different type contexts. Instead, constants and variables have fixed types just as they have fixed names: thus, the variable $f_{\iota\to\iota}$ has the name $f_{\iota\to\iota}$ and the type $\iota \to \iota$. This handling of type information is also called Church-style (as opposed to Curry-style). In this paper we often omit the type subscript if it can easily be inferred in the given context.

3.4. Formulas as terms of type o

In most presentations of first-order logic, terms can be components of formulas but formulas are never components of terms. Church's STT allows for this later possibility as well: formulas and logical connectives are allowed within terms. Such a possibility will greatly increase both the expressive strength of the logic and the complexities of automated reasoning in the logic. STT achieves this intertwining of terms and formulas by using the special primitive type o to denote those simply typed terms that are the formulas of STT. Thus, we introduce logical connectives as specific constant constructors of type o. Since Church's version of STT was based on classical logic, he chose for the primitive logical connectives $\neg_{o\to o}$ for negation, $\vee_{o\to o\to o}$ for disjunction, and for each type γ, a symbol $\forall_{(\gamma\to o)\to o}$. Other logical connectives, such as $\wedge_{o\to o\to o}$, $\supset_{o\to o\to o}$, and $\exists_{(\gamma\to o)\to o}$ (for every type γ), can be introduced by either also treating them as primitive or via definitions. A *formula* or *proposition* of STT is a simply typed λ-term of type o and a *sentence* of STT is a closed proposition (i.e., containing no free variables). In order to make the syntax of λ-terms converge more to the conventional syntax of logical formulas, we shall adopt the usual conventions for quantified expressions by using the abbreviations $\forall x_\gamma.B$ and $\exists x_\gamma.B$ for $\forall_{(\gamma\to o)\to o}\lambda x_\gamma.B$ and $\exists_{(\gamma\to o)\to o}\lambda x_\gamma.B$, respectively. Similarly, the familiar infix notion $B \vee C$, $B \wedge C$, and $B \supset C$ abbreviate the expressions $((\vee_{o\to o\to o}B)C)$, $((\wedge_{o\to o\to o}B)C)$, and $((\supset_{o\to o\to o} B)C)$, respectively.

Beyond the logical connectives and quantifiers of classical logic, STT also contains the constant $\iota_{(\gamma\to o)\to\gamma}$ for each simple type γ. This constant is axiomatized in STT so that $\iota_{(\gamma\to o)\to\gamma}B$ denotes a member of the set that is described

by the expression B of type $\gamma \to o$. Thus, $\iota_{(\gamma\to o)\to\gamma}$ is used variously as a description operator or a choice function depending on the precise set of axioms assumed for it. Choice selects a member from B if B is non-empty and description selects a member from B only if B is a singleton set.

3.5. Elementary type theory

Church (1940) gives a Frege-Hilbert style logical calculus for deriving formulas. The inference rules can be classified as follows.

I–III. One step of α-conversion, β-reduction, or β-expansion.
IV. Substitution: From $F_{\tau\to o}X_\tau$, infer $F_{\tau\to o}A_\tau$ if X is not free in F.
V. Modus Ponens: From $A \supset B$ and A, infer B.
VI. Generalization: From $F_{\tau\to o}X_\tau$, infer $\forall_{(\tau\to o)\to o}F_{\tau\to o}$ if X is not free in F.

In addition to the inference rules, Church gives various axiom schemas. Consider first the following axiom schemas.

1–4. Classical propositional axioms
5$^\tau$. For every simple type τ, $\forall_{(\tau\to o)\to o}F_{\tau\to o} \supset FX$.
6$^\tau$. For every simple type τ, $\forall X_\tau(p_o \vee F_{\tau\to o}X) \supset p \vee \forall_{(\tau\to o)\to o}F$.

These axioms (together with the inference rules above) describe the theorems of what is often called *elementary type theory* (ETT) (Andrews, 1974): these axioms simply describe an extension of first-order logic with quantification at all simple types and with the term structure upgraded to be all simply typed λ-terms. In the last century, much of the work on the automation of higher-order logic focused on the automation of the elementary type theory.

3.6. Simple type theory

In order to encode mathematical concepts, additional axioms are needed which, in turn, requires that we introduce expressions for denoting equality and natural numbers.

Equality. Equality for terms of type τ, $A_\tau \doteq^\tau B_\tau$, is defined using Leibniz's formula $\forall P_{\tau\to o}.PA \supset PB$. By $A_\tau \neq^\tau B_\tau$ we mean $\neg(A_\tau \doteq^\tau B_\tau)$.

Natural numbers. An individual natural number n is denoted by the *Church numeral* encoding n-fold iteration (Church, 1936). Thus, the following denote the λ-calculus encoding of zero, one, two and three (here, τ is any simple type).

$$\lambda f_{\tau\to\tau}\lambda x_\tau.x$$
$$\lambda f_{\tau\to\tau}\lambda x_\tau.fx$$
$$\lambda f_{\tau\to\tau}\lambda x_\tau.f(fx)$$
$$\lambda f_{\tau\to\tau}\lambda x_\tau.f(f(fx))$$

Notice that if we denote by $\hat{\tau}$ the type $(\tau \to \tau) \to \tau \to \tau$, then all these terms are of type $\hat{\tau}$. The λ-abstraction $\lambda n_{\hat{\tau}}\lambda f_{\tau\to\tau}\lambda x_\tau..f(nfx)$ is denoted $S_{\hat{\tau}\to\hat{\tau}}$ and

has the property that it computes the successor of a number encoded in this fashion. The set of all natural numbers (based on iteration of functions over type τ) can be defined as the λ-abstraction

$$\lambda n_{\hat{\tau}} \forall p_{\hat{\tau}\to o}.\, (p 0_{\hat{\tau}} \supset ((\forall x_{\hat{\tau}}.\, px \supset p(S_{\hat{\tau}\to\hat{\tau}} x)) \supset pn))$$

of type $\hat{\tau} \to o$. This expression uses higher-order quantification to define the set of all natural numbers as being the set of terms n that are members of all sets that contain zero and are closed under successor. It is unfortunate that the encoding of numbers is dependent on a specific type: in other words, there is a different set of natural numbers for every type τ. The polymorphic type system of Girard (1971, 1986) and Reynolds (1974) fixed this problem by admitting within λ-terms the ability to abstract and apply types.

Adding the following axioms to those of the elementary type theory yields Church's simple theory of types (STT).

7. There exists at least two individuals: $\exists X_\iota Y_\iota.\, X \neq^\iota Y$.

8. Infinity: The successor function on Church numerals at type $(\iota \to \iota) \to \iota \to \iota$ is injective.

9^τ. Description: $F_{\tau\to o} X_\tau \supset (\forall Y_\tau.\, FY \supset X \doteq Y) \supset F(\iota_{(\tau\to o)\to\tau} F)$.

$10^{\tau\to\gamma}$. Functional extensionality: $(\forall X_\tau.\, FX \doteq GX) \supset F \doteq^{\tau\to\gamma} G$.

11^τ. Choice: $F_{\tau\to o} X_\tau \supset F(\iota_{(\tau\to o)\to\tau} F)$.

Church also mentions the possibility of including an additional axiom of extensionality of the form $P \Leftrightarrow Q \supset P \doteq^o Q$. In fact, Henkin (1950) includes this axiom as part of his Axiom 10 (and he excludes axioms 7–9^τ). We follow Henkin and include the following axiom as part of Axiom 10.

10^o. Boolean extensionality: $P \Leftrightarrow Q \supset P \doteq^o Q$.

The description axioms (Axioms 9^τ) allow us to use $\iota_{(\tau\to o)\to\tau}$ to extract the unique element of any singleton set. If we assume the description axioms, then we can prove that every functional relation corresponds to a function. That is, we can prove

$$(\forall x_\tau \exists y_\gamma.\, r_{\tau\to\gamma\to o} xy \wedge \forall z_\gamma.\, (rxz \supset y \doteq z)) \supset \exists f_{\tau\to\gamma} \forall x_\tau.\, rx(fx).$$

This fact may be used to justify restricting the relational perspective for the foundation of mathematics since functions are derivable from relations in STT.

The choice axioms (Axioms 11^τ) are strictly stronger than the description axioms (see Andrews (1972a)), i.e., choice implies description. Many interactive theorem provers include a choice operator, but the systematic inclusion of choice (or description) into automated procedures has only happened recently (see also Section 6).

Finally, Axioms 7 and 8 guarantee that there will be infinitely many individuals. There are many ways to add an axiom of infinity and Church's choice is convenient for developing some basic number theory using Church numerals.

3.7. Variants of elementary and simple type theory

Besides the various subsets of STT that involve choosing different subsets of the axioms to consider, other important variants of ETT and STT have been developed.

Adding to ETT the axioms of Boolean and functional extensionality (Axioms 10^o and $10^{\tau\to\beta}$), and possibly choice, gives a theory sometimes called *extensional type theory* (ExTT): equivalently STT without description, infinity and axiom 7. This is the logic studied by Henkin (1950), and it is the logic that is automated by the theorem provers described in Section 7.3. One cannot prove from ETT alone that η-conversion preserves the equality of terms: a fact that is provable, however, using ExTT. Also, Boolean extensionality can be considered without including functional extensionality and vice versa. Most modern implementations of ETT generally treat the equality of typed λ-terms up to $\beta\eta$-conversion. By doing this some weak form of extensionality is thus automatically guaranteed. However, this should not be confused with supporting full functional and Boolean extensionality (cf. the discussion of non-functional models and extensionality in Section 4.1).

While Church axiomatized the logical connectives of STT in a rather conventional fashion (using, for example, negation, conjunction, and universal quantification as the primitive connectives), Henkin (1963) and Andrews (1972a, 2002) provided a formulation of STT in which the sole logical connective was equality (at all types). Not only was a formulation of logic using just this one logical connective perspicuous, it also improved on the semantics of Henkin's general models (Henkin, 1950).

Probably the most significant other variant of STT is one that replaces the classical logic underlying it with intuitionistic logic: several higher-order logic systems (e.g., Coq) are based on such a variant of STT. Intuitionistic variants of STT are easily achieved by changing the logical axioms of STT from those for classical logic to those for intuitionistic logic.

A logic of unity. ETT (and analogously ExTT or STT) provides a framework for considering propositional logic and first-order logic as fragments of higher-order logic. In the era of computer implementations of logic, this unifying aspect of ETT is of great value: an implementation of aspects of ETT immediately can be seen as an implementation that is also effective for these two kinds of simpler logics.

3.8. An example

Consider formalizing the most basic notions of point-set topology in STT. First, we formalize some simple set-theoretic concepts using the following typed constants and λ-expressions.

empty set	$\emptyset_{\iota\to o}$	$\lambda x.x \neq x$ (or $\lambda x.\bot$)
membership	$\in_{\tau\to(\tau\to o)\to o}$	$\lambda x\lambda C.\,Cx$
subset	$\subseteq_{(\iota\to o)\to(\iota\to o)\to o}$	$\lambda A\lambda B\forall x.\,Ax \supset Bx$
intersection	$\cap_{(\iota\to o)\to(\iota\to o)\to\iota\to o}$	$\lambda A\lambda B\lambda x.\,Ax \wedge Bx$
family union	$\bigcup_{((\iota\to o)\to o)\to\iota\to o}$	$\lambda D\lambda x\exists C.\,DC \wedge Cx$

We now define the symbol $open_{((\iota\to o)\to o)\to(\iota\to o)\to o}$ so that $(open\ \mathcal{C}\ S)$ holds when $\mathcal{C}$ is a topology (a collection of open sets) on S. Informally, this should hold when $\mathcal{C}$ is a set of subsets of S such that $\mathcal{C}$ contains the empty set as well as the set S and it is closed under (binary) intersections and arbitrary unions. Formally, the symbol *open* can be defined as the λ-abstraction

$$\lambda\mathcal{C}\lambda S.(\emptyset \in \mathcal{C}) \wedge (S \in \mathcal{C}) \wedge [\forall A \forall B.(A \in \mathcal{C} \wedge B \in \mathcal{C} \supset (A \cap B) \in \mathcal{C}] \\ \wedge [\forall \mathcal{B}.(\mathcal{B} \subseteq \mathcal{C}) \supset (\bigcup \mathcal{B}) \in \mathcal{C}]$$

A simple fact about open sets is the following. Assume that $\mathcal{C}$ is a topology for S. If G is a subset of S and all elements of G are also members of an open set (i.e., of a member of $\mathcal{C}$) that is a subset of G, then G itself is open. We can formalize this theorem as the following formula in STT.

$$\forall\mathcal{C}\forall S\forall G.\ (open\ \mathcal{C}\ S) \supset [\forall x.\ x \in G \supset \exists S.\ S \in \mathcal{C} \wedge x \in S \wedge S \subseteq G] \supset (G \in \mathcal{C})$$

This formula is provable in STT if we employ the functional extensionality axiom $10^{\iota\to o}$ in order to show that the two predicates G and

$$\bigcup(\lambda H.\ (open\ \mathcal{C}\ H) \wedge (H \subseteq G))$$

(both of type $\iota \to o$) are equal. Since it is an easy matter to prove that this second expression is in $\mathcal{C}$, Leibniz's definition of equality immediately concludes that G must also be in $\mathcal{C}$.

One weakness of using STT for formalizing an abstract notion of topology is that we provided above a definition in which open sets were sets of individuals: that is, they were of type $\iota \to o$. Of course, it might be interesting to consider topologies on other types, for example, on sets of sets. We could adopt the technique used in Church (1940) of indexing most notions with types, such as, for example, $\subseteq^{\tau}$. More expressive logics with richer treatment of types and their quantification are desirable: examples of such logics include Girard's System F (Girard, 1986), Reynold's polymorphic λ-calculus (Reynolds, 1974), and Andrews's transfinite type system (Andrews, 1965).

3.9. Church used different syntax not adopted here

Church's introduction of λ as a prefix operator to denote function abstraction and the use of juxtaposition to denote function application is now well established syntax. On the other hand, Church used a number of syntactic conventions and choices that appear rather odd to the modern reader. While Church used a simplification of the dot notation used in Whitehead and Russell (1910, 1912, 1913), most uses of dots in syntax have been dropped in modern systems, although a dot is sometimes retained to separate a bound variable from the body of a λ-term. Church similarly used concatenation to denote function types, but most modern systems use an arrow. The use of omicron as the type for propositions survives in some systems while many other systems use *Prop* (the latter is used more frequently in systems for ETT, while the former seems more prominent in systems for ExTT; see also the distinction between types

prop and *bool* in the Isabelle system; more provers for ETT and ExTT are presented in Section 7). Similarly, the connectives $\forall_{(\gamma\to o)\to o}$ and $\exists_{(\gamma\to o)\to o}$ are often replaced by the binders $\forall$ and $\exists$, respectively, although they are often used to denote quantification at the level of types in certain strong type systems.

4. Meta-theory

Church (1940) proved that the deduction theorem holds for the proof system consisting of the axioms and inference rules described in Section 3. The availability of the deduction theorem means that the familiar style of reasoning from assumptions is valid in STT. Church also proved a number of theorems regarding natural numbers and the possibility of defining functions using primitive recursive definitions. The consistency of STT and a formal model theory of STT were left open by Church.

4.1. Semantics and cut-elimination

We outline below several major meta-theoretic results concerning STT and closely related logics.

Standard models. Gödel's incompleteness theorem (Gödel, 1931) can be extended directly to ETT (and ExTT or STT) since second-order quantification can be used to define Peano arithmetic: that is, there is a "true" formula of ETT (or any extension of it) that is not provable. The notion of truth here, however, is that arising from what is called the *standard model* of ETT (resp. any extension of it) in which a functional type, say, $\gamma \to \tau$, contains *all* functions from the type γ to the type τ. Moreover, the type o is assumed to contain exactly two truth values, namely *true* and *false*.

Henkin models. Henkin (1950) introduced a broader notion of *general model* in which a type contains "enough" functions but not necessarily all functions. Henkin then showed soundness and completeness. More precisely, he showed that provability in ExTT coincides with truth in all general models (the standard one as well as the non-standard ones). Andrews (1972b) provided an improvement on Henkin's definition of general models by replacing the notion that there be enough functions to provide denotations for all formulas of ETT with a more direct means to define general models based on combinatory logic. Andrews (1972a) points out that Henkin's definition of general model technically was in error since his definition of general models admitted models in which the axiom of functional extensionality ($10^{\tau\to\beta}$) does not hold. Andrews then showed that there is a rather direct way to fix that problem by shifting the underlying logical connectives away from the usual Boolean connectives and quantifiers for a type-indexed family of connectives $\{Q_{\tau\to\tau\to o}\}_\tau$ in which $Q_{\tau\to\tau\to o}$ denotes equality at type τ.

Non-functional models and extensionality. Henkin models are fully extensional, i.e., they validate the functional and Boolean extensionality axioms $10^{\tau\to\gamma}$ and 10^{o}. The construction of non-functional models for ETT has been pioneered by Andrews (1971). In Andrews's so-called v-complexes, which are based on Schütte's semi-valuation method (Schütte, 1960), both the functional and the Boolean extensionality principles fail. Assuming β-equality, functional extensionality $10^{\tau\to\gamma}$ splits into two weaker and independent principles η ($F \doteq \lambda X.\, FX$, if X is not free in term F) and ξ (from $\forall X.F \doteq G$ infer $\lambda X.\, F \doteq \lambda X.\, G$, where X may occur free in F and G). Conversely, $\beta\eta$-conversion, which is built-in in many modern implementations of ETT, together with ξ implies functional extensionality. Boolean extensionality, however, is independent of any of these principles. A whole landscape of respective notions of models structures for ETT between Andrews's v-complexes and Henkin semantics that further illustrate and clarify the above connections is developed in Benzmüller et al. (2004); Brown (2004); Benzmüller (1999a), and an alternative development and discussion has been contributed by Muskens (2007).

Takeuti's conjecture. Takeuti (1953) defined GLC ("generalize logical calculus") by extending Gentzen's LK with (second-order) quantification over predicates. He conjectured cut-elimination for the GLC proof system and he showed that this conjecture proved the consistency of analysis (second-order arithmetic). Schütte (1960) presented a simplified version of Takeuti's GLC and gave a semantic characterization of the Takeuti conjecture. This important conjecture was proved by Tait (1966) for the second-order fragment using Schütte's semantic results. The higher-order version of the conjecture was later proved by Takahashi (1967) and by Prawitz (1968). The proof of strong normalization for System F given by Girard (1971) also proves Takeuti's conjecture as a consequence. Andrews (1971) used the completeness of cut-free proofs (but phrased in the contrapositive form as the *abstract consistency principle* (Smullyan, 1963)) in order to give a proof of the completeness of resolution in ETT. Takeuti (1975) presented a cut-free sequent calculus with extensionality that is complete for Henkin's general models. The abstract consistency proof technique, as used by Andrews, has been further extended and applied to obtain cut-elimination results for different systems between ETT and ExTT by Brown (2004), Benzmüller et al. (2004, 2008a), and Brown and Smolka (2010). For a different semantic approach to proving cut-elimination for intuitionistic variants of STT see Hermant and Lipton (2010).

Candidates of reducibility. In the setting of the intuitionistic variants of STT, the proofs themselves are of interest since they can be seen as programs that carry the computational content of constructive proofs. Girard (1971, 1986) proved the strong normalization of such proofs (expressed as richly typed λ-terms). To achieve this strong normalization result, Girard introduces the *candidats de reductibilité* technique which is, today, a common technique used to prove results such as cut-elimination for higher-order logics.

Herbrand's theorem for ETT. In Andrews et al. (1984), Andrews introduced a notion of proof called a *development* that resembles Craig-style linear reasoning in which a formula can be repeatedly rewritten until a tautologous formula is encountered. Three kinds of formula rewritings are possible: instantiate a top-level universal quantifier (with an eigenvariable), instantiate a top-level existential quantifier (with a term), or duplicate a top-level existential quantifier. Completeness of developments for ETT can be taken as a kind of Herbrand theorem for ETT. Miller (1983, 1987) presented the rewrites of developments as a tree instead of a line. The resulting proof structure, called *expansion trees*, provides a compact presentation of proofs for higher-order classical logic. Expansion trees are a natural generalization of Herbrand disjunctions to formulas which might not be in prenex normal form and where higher-order quantification might be involved.

4.2. Cut-simulation properties

Cut-elimination in first-order logic gives rise to the *subformula property*: that is, cut-free proofs are arrangements of formulas which are just subformulas of the formulas in the sequent at the root of the proof. In ETT (and ExTT or STT), however, cut-free proofs do not necessarily satisfy this subformula property. To better understand this situation remember that predicate variables may be instantiated with terms that introduce new formula structure. For this reason, the subformula property may break (cf. Section 6.1). However, at the same time this offers the opportunity to mimic cut-introductions by appropriately selecting such instantiations for predicate variables. For example, a cut formula φ may be introduced by instantiating the law of excluded middle $\forall P.P \vee \neg P$ with φ and by applying disjunction elimination (i.e., the rule of cases). In other words, one may trivially eliminate cut-rule applications by instead working with the axiom of excluded middle.[8] As shown by Benzmüller et al. (2009), effective cut-simulation is also supported by other prominent axioms, including comprehension, induction, extensionality, description, and choice. Also arbitrary (positive) Leibniz equations can be employed for the task.

Cut-simulations have in fact been extensively used in literature. For example, Takeuti showed that a conjecture of Gödel could be proved without cut by using the induction principle instead (Takeuti, 1960); McDowell and Miller (2002) illustrate how the induction rule can be used to hide the cut rule; and Schütte (1960) used excluded middle to similarly mask the cut rule.

In higher-order logic, cut-elimination and cut-simulation should always be considered in combination: a pure cut-elimination result may indeed mean little if at the same time axioms are assumed that support effective cut-simulation.

Church's use of the λ-calculus to build comprehension principles into the language can therefore be seen as a first step in the program to eliminate the need for cut-simulating axioms. Further steps have recently been achieved, and

[8] For automating higher-order logic it is thus very questionable to start with intuitionistic logic first and to simply add the law of excluded middle to arrive at classical logic.

tableaux and resolution calculi have been presented that employ primitive equality and which provide calculus rules (as opposed to an axiomatic treatment) for extensionality and choice (cf. Section 6.3). These calculus rules do not support cut-simulation.

4.3. Higher-order substitutions and normal forms

One of the challenges posed by higher-order substitution is that the many normal forms on which theorem provers often rely are not stable under such substitution. Clearly, a formula in $\beta\eta$-normal form may no longer be in $\beta\eta$-normal form after a λ-term instantiates a higher-order free variable in it. Similarly, many other normal forms—e.g., negation normal, conjunctive normal, and Skolem normal—are not preserved under such substitutions. In general, this instability is not a major problem since often one can re-normalize after performing such substitutions. For example, one often immediately places terms into $\beta\eta$-normal form after making a substitution. Since there can be an explosion in the size of terms when such normalization is made, there are compelling reasons to delay such normalization (Liang et al., 2005). Andrews (1971), for example, integrates the production of conjunctive normal and Skolem normal forms within the process of doing resolution.

4.4. Encodings of higher-order logic into first-order logic

Given the expressiveness of first-order logic and that theoremhood in both first-order logic and ETT (and ExTT or STT) is recursively enumerable, it is not a surprise that provability in the latter can be formalized in first-order logic. Some of the encodings have high-enough fidelity to make it possible to learn something structural about ETT from its encoding. For example, Dowek (2008) and Dowek et al. (2001) use an encoding of ETT in first-order logic along with Skolemization in first-order logic in order to explain the nature of Skolemization in ETT.

Mappings of second-order logic into many-sorted first-order logic have been studied by Enderton (1972). Henschen (1972) presents a mapping from higher-order logic and addresses the handling of comprehension axioms. For (type restricted) ExTT with Henkin-style semantics, complete translations into many-sorted, first-order logic have been studied by Kerber (1991, 1994).

Modern interactive theorem provers such as Isabelle nowadays employ translations from polymorphic higher-order logic into (unsorted or many-sorted) first-order logic in order to employ first-order theorem provers to help prove subgoals. Achieving Henkin completeness is thereby typically not a main issue. The focus is rather on practical effectiveness. Even soundness may be abandoned if other techniques, such as subsequent proof reconstruction, can be employed to identify unsound proofs. Relevant related work has been presented by Hurd (2003), Meng and Paulson (2008), and Blanchette et al. (2013b).

5. Skolemization and unification

In the latter sections of this chapter, we describe a number of theorem provers for various subsets of STT. They all achieve elements of their automation in ways that resemble provers in first-order logic. In particular, when quantifiers are encountered, they are either instantiated with *eigenvariables* (in the sense of Gentzen (1969a)) or, dually, instantiated by new free variables called *logic variables*: such variables denote a term that is determined later via *unification*. To simplify the relationship between eigenvariables and logic variables and, hence, simplify the implementation of unification, it is customary to simplify quantifiers prior to performing proof search. In classical first-order logic theorem provers, *Skolemization* provides such simplification and unification does not need to deal with eigenvariables at all.

While such a simplification of quantificational structure is possible in classical higher-order theorem provers, some important issues arise concerning quantifier alternation, Skolemization, and term unification that are not genuine issues in a first-order setting. We discuss these differences below.

5.1. Skolemization

A typical approach to simplifying the alternation of quantifiers in first-order logic is to use Skolemization. Such a technique replaces an assumption of the form, say, $\forall x_\tau \exists y_\delta . Pxy$ with the assumption $\forall x_\tau . Px(fx)$, where f is a new constant of type $\tau \to \delta$. The original assumption is satisfiable if and only if the Skolemized formula is satisfiable: in a model of the Skolemized formula, the meaning of the *Skolem function* f is a suitable choice function.

Lifting Skolemization into higher-order logic is problematic for a number of reasons. First, for a logic such as ETT which does not accept the axiom of choice, Skolem functions should not be allowed, at least not without restrictions. For example, the resolution system for ETT introduced by Andrews (1971) used Skolem functions to simplify quantifier alternations. While Andrews was able to prove that resolution was complete for ETT, he did not provide the converse result of soundness since some versions of the axiom of choice could be proved (Andrews, 1973). As was shown by Miller (1983, 1992), the soundness of Skolem functions can be guaranteed by placing suitable restrictions on the occurrences of Skolem functions within λ-terms. In particular, consider an assumption of the form $\forall x_\tau \exists y_{\delta \to \theta} . Pxy$ and its Skolemized version $\forall x_\tau . Px(fx)$, where f is a new Skolem function of type $\tau \to \delta \to \theta$. In order for a proof not to "internalize" the choice function named by f, every substitution term t used in that proof must be restricted so that every occurrence of f in t must have at least one argument and any free variable occurrences in that argument must also be free in t. Thus it is not possible to form an abstraction involving the Skolemization-induced argument and, in that way, the Skolem function is not used as a general choice function.

A second problem with using Skolemization is that there are situations where a type may have zero or one inhabitant prior to Skolemization but can have an infinite number of inhabitants after Skolemization (Miller, 1992). Such a

change in the nature and number of terms that appear in types before and after Skolemization introduces new constants is a serious problem when a prover wishes to present its proofs in forms that do not use Skolemization (such as natural deduction or sequent calculus).

A third problem using Skolemization is that in the unification of typed λ-terms, the treatment of λ-abstractions and the treatment of eigenvariables are intimately related. For example, the unification problems $\exists w_\iota.(\lambda x_\iota.x) = (\lambda x_\iota.w)$ and $\exists w_\iota \forall x_\iota.x = w$ are essentially the same: since the second (purely first-order) problem is not provable in first-order logic (since it is true only in a singleton domain), the original unification problem also has no solutions. Explaining the non-unification of terms $\lambda x_\iota.x$ and $\lambda x_\iota.w$ in terms of Skolemization and choice functions seems rather indirect.

5.2. Unification of simply typed λ-terms

Traditionally, the unification of simply typed λ-terms can be described as proving the formula

$$\exists x^1_{\tau_1} \ldots \exists x^n_{\tau_n}.\ t_1 = s_1 \wedge \cdots \wedge t_m = s_m \quad (n, m \geq 0).$$

If we make the additional assumption that no variable in the quantifier prefix is free in any of the terms $s_1, \ldots, s_m$ then this formula is also called a *matching problem*. The order of the unification problem displayed above is $1 + \max\{ord(\tau_1), \ldots, ord(\tau_n)\}$; thus, if $n = 0$ that order is 1. Andrews showed (Andrews, 1974, Theorem 2) that such a formula is provable in ETT if and only if there is a substitution θ for the variables in the quantifier prefix such that for each $i = 1, \ldots, n$, the terms $t_\iota\theta$ and $s_\iota\theta$ have the same normal form. Such a substitution as θ is called a *unifier* for that unification problem. Such unification problems have been studied in which the common normal form is computed using just β-conversion or with $\beta\eta$-conversion: thus one speaks of unification or matching modulo β or modulo $\beta\eta$. This theorem immediately generalizes a similar theorem for first-order logic.

Although Guard and his student Gould investigated higher-order versions of unification as early as 1964 (Guard, 1964; Gould, 1966), it was not until 1972 that the undecidability of such unification was demonstrated independently by Huet (1973a) and Lucchesi (1972). Those two papers showed that third-order unification was undecidable; later Goldfarb (1981) showed that second-order unification was also undecidable. The decidability of higher-order matching was shown after several decades of effort: it was first shown for second-order matching in Huet and Lang (1978); for third-order matching in Dowek (1992); and for fourth-order matching in Padovani (2000). Finally, Stirling (2009) has shown that matching at all orders is decidable.

Following such undecidability results for unification, the search for unification procedures for simply typed λ-terms focused on the recursive enumeration of unifiers. The first such enumeration was presented in (Pietrzykowski and Jensen, 1972; Pietrzykowski, 1973; Jensen and Pietrzykowski, 1976). Their enumeration was intractable in implemented systems since when it enumerated a

unifier, subsequent unifiers in the enumeration would often subsume it, thus leading to a highly redundant search for unifiers.

Huet (1975) presented a different approach to the enumeration of unifiers. Instead of solving all unification problems, some unification pairs (the so-called flex-flex pairs) were deemed too unconstrained to schedule for solving. In such problems, the head of all terms in all equalities are existentially quantified. For example, the unification problem $\exists f_{\iota\to\iota}\exists g_{\iota\to\iota}.fa = ga$ is composed of only flex-flex pairs and it has a surprising number of unifiers. In particular, let t be any $\beta\eta$-normal closed term of type ι and assume that the constant a has n occurrences in t. There are 2^n different ways to abstract a from t and by assigning one of these to f and possibly another to g we have a unifier for this unification problem. Clearly, picking blindly from this exponential set of choices on an *arbitrary* term is not a good idea. An important design choice in the semi-decision procedure of Huet (1975) is the delay of such unification problems. In particular, Huet's procedure computed "pre-unifiers"; that is, substitutions that can reduce the original unification problem to one involving only flex-flex equations. Huet showed that the search for pre-unifiers could be done, in fact, without redundancy. He also showed how to build a resolution procedure for ETT on pre-unification instead of unification by making flex-flex equations into "constraints" on resolution (Huet, 1972, 1973b). The earliest theorem provers for various supersets of ETT—TPS (Andrews et al., 1996), Isabelle (Paulson, 1989), and λProlog (Nadathur and Miller, 1988; Miller and Nadathur, 2012)—all implemented rather directly Huet's search procedure for pre-unifiers.

The unification of simply typed λ-terms does not have the most-general-unifier property: that is, there can be two unifiers and neither is an instance of the other. Let g be a constant of type $\iota \to \iota \to \iota$ and a a constant of type ι. Then the second-order unification problem $\exists f_{\iota\to\iota}.fa = gaa$ has four unifiers in which f is instantiated with $\lambda w.gww$, $\lambda w.gwa$, $\lambda w.gaw$, and $\lambda w.gaa$. A theorem prover that encounters such a unification problem may need to explore all four of these unifiers during the search for a proof. It is also possible for a unification problem to have an infinite number of unifiers that are not instances of one another. Such is the case for the unification problem $\exists f_{\iota\to\iota}.\lambda x.f(hx) = \lambda x.h(fx)$, where h is a constant of type $\iota \to \iota$. All the following instantiations for f yield a unifier: $\lambda w.w$, $\lambda w.hw$, $\lambda w.h(hw)$, $\lambda w.h(h(hw))$, ...

For more details about Huet's search procedure for unifiers, we recommend Huet's original paper (Huet, 1975) as well as the subsequent papers by Snyder and Gallier (1989) and Miller (1992), and the handbook chapter by Dowek (2001). Here we illustrate some of the complexities involved with this style of unification.

5.3. Mixed prefix unification problems

As we motivated above, it is natural to generalize unification problems away from a purely existential quantifier prefix to one that has a *mixed quantifier*

prefix, i.e., a *unification problem* will be a formula of the form

$$Q_1 x^1_{\tau_1} \ldots Q_n x^n_{\tau_n}.\ t_1 = s_1 \wedge \cdots \wedge t_m = s_m \quad (n, m \geq 0).$$

Here, Q_ι is either $\forall$ or $\exists$ for $i = 1, \ldots, n$. There is, in fact, a simple technique available in higher-order logic that is not available in first-order logic which can simplify quantifier alternation in such unification problems. In particular, if $\forall x_\tau \exists y_\sigma$ occurs within the prefix of a unification problem, it is a simple matter to "rotate" the $\forall x$ to the right: this requires "raising" the type of the $\exists y$ quantifier. That is, $\forall x_\tau \exists y_\sigma$ can be replaced by $\exists h_{\tau\to\sigma} \forall x_\tau$ if all occurrences of y in the scope of $\exists y$ are substituted by (hx). The resulting two unification problems are equivalent in the sense that unifiers for these two problems can be put into a one-to-one correspondence by a simple mapping. For example, the unification problem $\forall x_\iota \forall y_\iota \exists z_\iota . fzx = fyz$ (for some constant f of type $\iota \to \iota \to \iota$) can be rewritten to the unification problem

$$\exists h_{\iota\to\iota\to\iota} \forall x_\iota \forall y_\iota . f(hxy)x = fy(hxy).$$

This latter problem can be replaced by the equivalent unification problem $\exists h_{\iota\to\iota\to\iota} . \lambda x \lambda y . f(hxy)x = \lambda x \lambda y . fy(hxy)$. Using the technique of raising, any unification problem with a mixed quantifier prefix can be rewritten to one with a prefix of the form $\exists\forall$. Furthermore, the block of $\forall$ quantifiers can be removed from the prefix if they are converted to a block of λ-bindings in front of all terms in all the equations. In this way, a mixed prefix can be rewritten to an equivalent one involving only existential quantifiers. Details of performing unification under a mixed prefix can be found in Miller (1992). The notion of $\forall$-lifting employed by the Isabelle prover can be explained using raising (Miller, 1991; Paulson, 1989).

5.4. Pattern unification

There is a small subset of unification problems, first studied by Miller (1991), whose identification has been important for the construction of practical systems. Call a unification problem a *pattern unification* problem if every occurrence of an existentially quantified variable, say, M, in the prefix is applied to a list of arguments that are all distinct variables bound by either a λ-binder or a universal quantifier in the scope of the existential quantifier. Thus, existentially quantified variables cannot be applied to general terms but a very restricted set of bound variables. For example,

$$\exists M \exists N . \lambda x \lambda y . f(Mxy) = \lambda x \lambda y . Ny \qquad \exists M \forall x \forall y . f(Mxy) = fy$$

$$\exists M \forall x . \lambda y . Mxy = \lambda y . Myx \qquad \exists M \exists N . \forall x \forall y . Mxy = Ny$$

are all pattern unification problems. All these unification problems have most general unifiers, respectively, $[M \mapsto \lambda x \lambda y . Py, N \mapsto \lambda y . f(Py)]$, $[M \mapsto \lambda x \lambda y . y]$, $[M \mapsto \lambda x \lambda y . P]$, and $[M \mapsto \lambda x \lambda y . Ny]$, where P is a new (existentially quantified) variable. Notice that although the last two of these are examples of flex-flex

unification problems, they both have a most general unifier. The following unification problems do not fall into this fragment:

$$\exists M \exists N. \lambda x. f(Mxx) = Nx \qquad \exists M. \forall x. f(Mx) = M(fx).$$

Notice that all first-order unification problems are, in fact, pattern unification problems, and that pattern unification problems are stable under the raising technique mentioned earlier. The main result about pattern unification is that—like first-order unification—deciding unifiability is decidable and most general unifiers exist for solvable problems. Also like first-order unification, types attributed to constructors are not needed for doing the unification.

5.5. Practical considerations

Earlier we mentioned that unification problems can be addressed using either just β-conversion or $\beta\eta$-conversion. Although Huet (1975) considered both unification modulo β and $\beta\eta$ conversion separately, almost no implemented system considers only the pure β conversion rules alone: term equality for STT is uniformly treated as $\beta\eta$-convertibility.

Skolemization is a common technique for simplifying quantifier alternation in many implemented higher-order theorem provers (cf. Section 7). On the other hand, several other systems, particularly those based on the intuitionistic fragment of ETT, do not use Skolemization: instead they either use raising, as is done in Isabelle (Paulson, 1989, 1994) or they work directly with a representation of an unaltered quantifier prefix, as is done in the Teyjus implementation (Nadathur and Linnell, 2005) of λProlog.

It is frequently the case that in computational logic systems that unify simply typed λ-terms, only pattern unification problems need to be solved. As a result, some systems—such as the Teyjus implementation of λProlog and the interactive theorem provers Minlog (Benl et al., 1998) and Abella (Gacek et al., 2012)—only implement the pattern fragment since this makes their design and implementation easier.

6. Challenges for automation

While theorem provers for ETT, ExTT, and STT can borrow many techniques from theorem provers for first-order logic, there are several challenges to the direct implementation of such provers. We discuss some of these challenges below.

6.1. Instantiation of predicate variables

During the search for proofs in quantificational logics, quantifiers need to be instantiated (possibly more than once) with various terms. Choosing such terms is a challenge partly because when a quantifier needs to be instantiated, the role of that instantiation term in later proof steps is not usually known. To address this gap between when a quantifier needs an instantiation term and when that

term's structure is finally relevant to the completion of a proof, the techniques of unification described in the previous section are used. When unification is involved, quantifiers are instantiated not with terms but with variables which represent a promise: before a proof is complete, those variables will be replaced by terms. The variables that are introduced in this way are sometimes called *logic variables*: these variables correspond to those marked using existential quantification in the unification problems of Section 5.3.

In this way, one can delay the choice of which term to use to instantiate the quantifier until the point where that term is actually used in the proof. As an illustration of using unification in the search for a proof, consider attempting a proof of the formula $(q\ (f\ a))$ from the conjunctive assumption

$$pa \wedge (\forall x.px \supset p(fx)) \wedge (\forall y.py \supset qy).$$

One way to prove this goal would be to assume, for example, that each universally quantified premise is used once for some, currently, unspecified term. In this case, instantiate $\forall x$ and $\forall y$ with logic variables X and Y, respectively, and we have an assumption of the form

$$pa \wedge (pX \supset p(fX)) \wedge (pY \supset qY).$$

We can then observe that the proof is complete if we chain together two applications of modus ponens: for that to work, we need to find substitution terms for X and Y to solve the equations

$$pa = pX \wedge p(fX) = pY \wedge qY = q(fa).$$

Clearly, this unification problem is solvable when X and Y are replaced by a and fa, respectively. Thus, if we were to repeat the steps of the proof but this time instantiate the quantifiers $\forall x$ and $\forall y$ with a and $(f\ a)$, respectively, the chaining of the modus ponens steps would now lead to a proper proof.

A key property of *first-order* quantificational logic is that the terms needed for instantiating quantifiers can all be found using unification of atomic formulas. When predicate variables are present, however, the unification of atomic formulas is no longer sufficient to generate all quantifier instantiations needed for proofs. For example, the ETT theorem

$$\exists p.(px \supset (ax \wedge bx)) \wedge ((ax \wedge bx) \supset px).$$

is proved by instantiating p with $\lambda w.aw \wedge bw$. If that quantifier were, instead, instantiated by the logic variable P to yield the formula

$$(Px \supset (ax \wedge bx)) \wedge ((ax \wedge bx) \supset Px)$$

no equalities between occurrences of atomic formulas will provide a unification problem that has this unifier. Similarly, the theorem

$$\forall q.(qa \supset qb) \supset pb \supset pa$$

is proved (in intuitionistic and classical logic) by instantiating $\forall q$ with the term $\lambda w.pw \supset pa$. Once again, however, if the quantifier $\forall q$ was instantiated with a logic variable Q, then no unification of that atomic formulas Qa, Qb, pa, and pb would have yielded this substitution terms for Q.

Of course, it is not surprising that simple syntactic checks involving subformulas are not sophisticated enough to compute substitutions for predicates. Often the key insight into proving a mathematical theorem is the production of the right set or relation to instantiate a predicate variable: in ETT (ExTT or STT), these would be encoded as λ-abstractions containing logical connectives. Similarly, induction can be encoded and the invariants for inductive proofs would be encoded as similar terms and used to instantiate predicate quantifiers. Nonetheless, a number of researchers have described various schemes for inventing substitution terms for predicate variables. We mention a few below.

Enumeration of substitutions. An early approach at the generation of predicate substitutions was provided by Huet (1972; 1973b) by essentially providing a mechanism for guessing the top-level, logical structure of a substitution for a predicate variable. Such guessing (called *splittings* in that paper) was interleaved with resolution steps by a system of constraints. Thus, his system suggested a candidate top-level connective for the body of a predicate substitution and then proceeded with the proof under that assumption.

A simple, prominent example to illustrate the need for splittings is $\exists P_o.\, P$. When using resolution the formula is first negated and then normalized to clause $\neg X_o$, where X is a predicate variable. There is no resolution partner for this clause available, hence the empty clause can not be derived. However, when guessing some top-level, logical structure for X, here the substitution $[\neg Y/X]$ is suitable, then $\neg\neg Y$ is derived, which normalizes into a new clause Y. Now, resolution between the clauses $\neg X$ and Y with substitution $[Y/X]$ directly leads to the empty clause.

Andrews's primitive substitutions (Andrews, 1989) incorporates Huet's notion of splitting, and an alternative description of splitting can be found in Dowek (1993).

Maximal set variables and set constraints. Bledsoe (1979) suggested a different strategy for coming up with predicate substitutions: in some cases, one can tell the maximal set that can solve a subgoal. Consider, for example, the formula

$$\exists A.\, (\forall x.Ax \supset px) \wedge \mathcal{C}(A)$$

Clearly there are many instantiations possible for A that will satisfy the first conjunct. For example, the empty set $\lambda w.\, \bot$, is one of them but it seems not to be the best one. Rather, a more appropriate substitution for A might be $\lambda w.\, pw \wedge Bw$, where B is a new variable that has the same type as A. This extension of the latter expression can then range from the empty set (where B is substituted by $\lambda w.\, \bot$) to $\lambda w.\, pw$ (where B is substituted by $\lambda w.\, \top$). Felty (2000) generalized and applied Bledsoe's technique, which was restricted to a subset of second-order logic, to the higher-order logic found in the calculus

of constructions. Moreover, Brown (2002) generalized Bledsoe's technique to ExTT. His solution, which employs reasoning with set constraints, has been applied within the TPS theorem prover.

6.2. Induction invariants

Induction and, to a lesser extent, co-induction are important inference rules in computer science and mathematics. Most forms of the induction rule require showing that some set is, in fact, an invariant of the induction. Even if one is only interested in first-order logic, the induction rule in this form requires discovering the instantiation of the predicate variable that codes the induction invariant. While Bledsoe (1979) provides some weak approaches to generating such invariants, a range of techniques are routinely used to provide either invariants explicitly or some evidence that an invariant exists. For an example of the latter, the work on cyclic proofs (Spenger and Dams, 2003) attempts to identify cycles in the unfolding of a proof attempt as a guarantee that an invariant exists. *Descente infinie* (sometimes also called *inductionless induction*) and *proof by consistency* (Comon, 2001) are also methods for proving inductive theorems without explicitly needing to invent an invariant (cf. also Wirth (2004)).

6.3. Equality, extensionality, and choice

There has been work on automating various axioms beyond those included in ETT. As mentioned above, various works have focused on automation of induction, which is based roughly on axioms 7 and 8. For many applications, including mathematics, one certainly wants and needs to have extensionality and maybe also choice (or description). However, the idea to treat such principles axiomatically, as e.g., proposed in Huet (1973b) for extensionality, leads to a significant increase of the search space, since these axioms (just like the induction axiom) introduce predicate variables and support cut-simulation (cf. Section 4.2). Another challenge is that unification modulo Boolean extensionality subsumes theorem proving: proving a proposition φ is the same as unifying φ and $\top$ modulo Boolean extensionality. More information on these challenges is provided by Benzmüller et al. (2009) and Benzmüller (2002).

Significant progress in the automation of ExTT in existing prover implementations has therefore been achieved after providing calculus level support for extensionality and also choice. Respective extensionality rules have been provided for resolution (Benzmüller, 1999b), expansion and sequent calculi (Brown, 2004, 2005), and tableaux (Brown and Smolka, 2010). Similarly, choice rules have been proposed for the various settings: sequent calculus (Mints, 1999), tableaux (Backes and Brown, 2011) and resolution (Benzmüller and Sultana, 2013).

Analogously, (positive) Leibniz equations are toxic for proof automation, since they also support cut-simulation. For this reason, the automation oriented tableaux and resolution approaches above support primitive equality and provide respective rules. The use of Leibniz equations can hence be omitted in the modeling of theories and conjectures in these approaches.

7. Automated theorem provers

7.1. Early systems

Probably the earliest project to mention is Peter Andrews NSF grant *Proof procedures for Type Theory* (1965-67). The goal was to lift ideas from propositional and first-order logic to the higher-order case. Also J. A. Robinson (1969, 1970) argued for the construction of automated tools for higher-order logic. Together with E. Cohen, Andrews started a first computer implementation based on the inference rules of Andrews (1971) and the unification algorithm of Huet (1975) in a subsequent project (1971-76). In 1977 this system did prove Cantor's theorem automatically in 259 seconds (Andrews and Cohen, 1977). After 1980, when D. Miller, F. Pfenning, and other students got involved, this theorem prover got substantially revised. The revised system was then called TPS. The TPS proof assistant (Miller et al., 1982; Andrews et al., 1996, 2000; Andrews and Brown, 2006), was, in fact, not based on resolution but on matrix-style theorem proving. Both λProlog (Nadathur and Miller, 1988) and the Isabelle theorem prover (Paulson, 1989) were early systems that implemented sizable fragments of the intuitionistic variants of ETT: they were tractable systems because they either removed or greatly restricted predicate quantification. Below we survey other higher-order systems that attempted to deal with interactive and automatic theorem proving in the presence of predicate quantification.

HOL. The ML based provers of the HOL family include HOL88, HOL98, and HOL4 (Gordon and Melham, 1993). These systems are all based on the LCF approach (Gordon et al., 1979), in which powerful proof tactics are iteratively built up from a small kernel of basic proof rules. Other LCF-based provers for higher-order logic are the minimalist system HOL Light (Harrison, 2009), which provides powerful automation tactics and which has recently played a key role in the verification of Kepler's conjecture (Hales, 2013), and the ProofPower system (Arthan, 2011), which provides special support for a set-theoretic specification language.

Isabelle/HOL. Isabelle (Paulson, 1989) is a theorem prover with a core tactic language built on a fragment of the intuitionistic variant of ETT. Built on this core is the Isabelle/HOL (Nipkow et al., 2002) interactive theorem prover for classical higher-order logic. Isabelle/HOL includes several powerful features such as bridges to external theorem provers, sophisticated user interaction, and the possibility to export executable specifications written in Isabelle/HOL as executable code in various programming languages.

PVS. The prototype verification system PVS (Owre et al., 1992) combines a higher-order specification languages with an interactive theorem proving environment that integrates decision procedures, a model checker, and various other utilities to improve user productivity in large formalization and verification projects. Like Isabelle and the HOL provers, PVS also includes a rich library of formalized theories.

IMPS. The higher-order interactive proof assistant IMPS (Farmer, 1993) provides good support for partial functions and undefined terms in STT (Farmer, 1990). Moreover, it supports human oriented formal proofs which are nevertheless machine checked. Most importantly, IMPS organizes mathematics using the "little theories" method in which reasoning is distributed over a network of theories linked by theory morphisms (Farmer et al., 1992). It is the first theorem proving system to employ this approach.

ΩMEGA. The higher-order proof assistant ΩMEGA (Benzmüller et al., 1997) combines tactic based interactive theorem proving with automated proof planning. With support from an agent-based model, it integrates various external reasoners: including first-order automated theorem provers, the higher-order automated theorem provers LEO (Benzmüller and Kohlhase, 1998; Benzmüller, 1999a) and TPS, and computer algebra systems (Autexier et al., 2010). Proof certificates from these external systems can be transformed and verified in ΩMEGA.

λClam and IsaPlanner. λ-Clam (Richardson et al., 1998) is a higher-order variant of the CLAM proof planner (Bundy et al., 1990) built in λProlog. This prover focuses on induction proofs based on the rippling technique. IsaPlanner (Dixon and Fleuriot, 2003) is a related generic proof planner built on top of the Isabelle system.

Deduction Modulo. In the deduction-modulo approach to theorem proving (Dowek et al., 2003), a first-order presentation of (intensional) higher-order logic can be exploited to automate higher-order reasoning (Dowek et al., 2001). A recent implementation of the deduction modulo approach (still restricted to first-order) has been presented by Burel (2011b); see also the Dedukti proof checker (Boespflug et al., 2012).

Other early interactive proof assistants, for variants of constructive higher-order logic, include Automath (Nederpelt et al., 1994), Nuprl (Constable et al., 1986), LEGO (Pollack, 1994), Coq (Bertot and Casteran, 2004), and Agda (Coquand and Coquand, 1999). The logical frameworks Elf (Pfenning, 1994), Twelf (Pfenning and Schürmann, 1999), and Beluga (Pientka and Dunfield, 2010) are based on dependently typed higher-order logic. Related provers include the general-purpose, interactive, type-free, equational higher-order theorem prover Watson (Holmes and Alves-Foss, 2001) and the fully automated theorem prover Otter-λ (Beeson, 2006) for λ-logic (a combination of λ-calculus and first-order logic). Abella (Gacek et al., 2012) is a recently implemented interactive theorem prover for an intuitionistic, predicative higher-order logic with inference rules for induction and co-induction. ACL2 (Kaufmann and Moore, 1997) and KeY (Beckert et al., 2007) are prominent first-order interactive proof assistants that integrate induction.

7.2. The TPTP THF initiative

To foster the systematic development and improvement of higher-order automated theorem proving systems, Sutcliffe and Benzmüller (2010), supported by several other members of the community, initiated the TPTP THF infrastructure (THF stands for *typed higher-order form*). This project has introduced the THF syntax for higher-order logic, it has developed a library of benchmark and example problems, and it provides various support tools for the new THF0 language fragment. The THF0 language supports ExTT (with choice) as also studied by Henkin (1950), that is, it addresses the most commonly used and accepted aspects of Church's type theory.

Version 6.0.0 of the TPTP library contains more than 3000 problems in the THF0 language.

The library also includes the entire problem library of Andrews's TPS project, which, among others, contains formalizations of many theorems of his textbook (Andrews, 2002). The first-order TPTP infrastructure (Sutcliffe, 2009) provides a range of resources to support usage of the TPTP problem library. Many of these resources are now immediately applicable to the higher-order setting although some have required changes to support the new features of THF. The development of the THF0 language, has been paralleled and significantly influenced by the development of the LEO-II prover (Benzmüller et al., 2008b). Several other provers have quickly adopted this language, leading to fruitful mutual comparisons and evaluations. Several implementation bugs in different systems have been detected this way.

7.3. TPTP THF0 compliant higher-order theorem provers

We briefly describe the currently available, fully automated theorem provers for ExTT (with choice). These systems all support the new THF0 language and they can be employed online (avoiding local installations) via Sutcliffe's SystemOnTPTP facility.[9]

TPS. The TPS prover can be used to prove theorems of ETT or ExTT automatically, interactively, or semi-automatically. When searching for a proof automatically, TPS first searches for an expansion proof (Miller, 1987) or an extensional expansion proof (Brown, 2004) of the theorem. Part of this process involves searching for acceptable matings (Andrews, 1981). Using higher-order unification, a pair of occurrences of subformulas (which are usually literals) is mated appropriately on each vertical path through an expanded form of the theorem to be proved. Skolemization and pre-unification is employed, and calculus rules for extensionality reasoning are provided. The behavior of TPS is controlled by sets of flags, also called modes. About fifty modes have been found that collectively suffice for automatically proving virtually all the theorems that

[9] See `http://www.cs.miami.edu/~tptp/cgi-bin/SystemOnTPTP`

TPS has proved automatically thus far. A simple scheduling mechanism is employed in TPS to sequentially run these modes for a limited amount of time. The resulting fully automated system is called *TPS (TPTP)*.

LEO-II. (Benzmüller et al., 2008b), the successor of LEO, is an automated theorem prover for ExTT (with choice) which is based on extensional higher-order resolution. More precisely, LEO-II employs a refinement of extensional higher-order RUE resolution (Benzmüller, 1999b). LEO-II employs Skolemization, (extensional) pre-unification, and calculus rules for extensionality and choice are provided. LEO-II is designed to cooperate with specialist systems for fragments of higher-order logic. By default, LEO-II cooperates with the first-order prover systems E (Schulz, 2002). LEO-II is often too weak to find a refutation among the steadily growing set of clauses on its own. However, some of the clauses in LEO-II's search space attain a special status: they are first-order clauses modulo the application of an appropriate transformation function. Therefore, LEO-II regularly launches time limited calls with these clauses to a first-order theorem prover, and when the first-order prover reports a refutation, LEO-II also terminates. Communication between LEO-II and the cooperating first-order theorem prover uses the TPTP language and standards. LEO-II outputs proofs in TPTP TSTP syntax.

Isabelle/HOL. The Isabelle/HOL system has originally been designed as an interactive prover. However, in order to ease user interaction several automatic proof tactics have been added over the years. By appropriately scheduling a subset of these proof tactics, some of which are quite powerful, Isabelle/HOL has in recent years been turned also into an automatic theorem prover, that can be run from a command shell like other provers. The latest releases of this automated version of Isabelle/HOL provide native support for different TPTP syntax formats, including THF0. The most powerful proof tactics that are scheduled by Isabelle/HOL include the *sledgehammer* tool (Blanchette et al., 2013a), which invokes a sequence of external first-order and higher-order theorem provers, the model finder *Nitpick* (Blanchette and Nipkow, 2010), the equational reasoner *simp* (Nipkow, 1989), the untyped tableau prover *blast* (Paulson, 1999), the simplifier and classical reasoners *auto*, *force*, and *fast* (Paulson, 1994), and the best-first search procedure *best*. The TPTP incarnation of Isabelle/HOL does not yet output proof terms.

Satallax. The higher-order, automated theorem prover Satallax (Brown, 2012, 2013) comes with model finding capabilities. The system is based on a complete ground tableau calculus for ExTT (with choice) (Backes and Brown, 2011). An initial tableau branch is formed from the assumptions of a conjecture and negation of its conclusion. From that point on, Satallax tries to determine unsatisfiability or satisfiability of this branch. Satallax progressively generates higher-order formulas and corresponding propositional clauses. Satallax uses the SAT solver MiniSat as an engine to test the current set of propositional clauses for unsatisfiability. If the clauses are unsatisfiable, the original branch is

unsatisfiable. Satallax employs restricted instantiation and pre-unification, and it provides calculus rules for extensionality and choice. If there are no quantifiers at function types, the generation of higher-order formulas and corresponding clauses may terminate. In that case, if MiniSat reports the final set of clauses as satisfiable, then the original set of higher-order formulas is satisfiable (by a standard model in which all types are interpreted as finite sets). Satallax outputs proofs in different formats, including Coq proof scripts and Coq proof terms.

Nitpick and Refute. These systems are (counter-)model finders for ExTT. The ability of Isabelle to find (counter-)models using the *Refute* and *Nitpick* (Blanchette and Nipkow, 2010) commands has also been integrated into automatic systems. They provide the capability to find models for THF0 formulas, which confirm the satisfiability of axiom sets, or the unsatisfiability of non-theorems. The generation of models is particularly useful for exposing errors in some THF0 problem encodings, and revealing bugs in the THF0 theorem provers. Nitpick employs Skolemization.

agsyHOL. The agsyHOL prover (Lindblad, 2013) is based on a generic lazy narrowing proof search algorithm. Backtracking is employed and a comparably small search state is maintained. The prover outputs proof terms in sequent style which can be verified in the Agda system.

coqATP. The coqATP prover (Bertot and Casteran, 2004) implements (the non-inductive) part of the calculus of constructions. The system outputs proof terms which are accepted as proofs by Coq (after the addition of a few definitions). The prover has axioms for functional extensionality, choice, and excluded middle. Propositional extensionality is not supported yet. In addition to axioms, a small library of basic lemmas is employed.

7.4. Recent applications of automated THF0 provers

Over the years, the proof assistants from Section 7.1 have been applied in a wide range of applications, including mathematics and formal verification. Typically these applications combine user interaction and partial proof automation. For further information we refer to the websites of these systems.

With respect to full proof automation the TPS system has long been the leading system, and the system has been employed to build up the TPS library of formalized and automated mathematical proofs. More recently, however, TPS is outperformed by several other THF0 theorem provers. Below we briefly point to some selected recent applications of the leading systems.

Both Isabelle/HOL and Nitpick have been successfully employed to check a formalization of a C++ memory model against various concurrent programs written in C++ (such as a simple locking algorithm) (Blanchette et al., 2011). Moreover, Nitpick has been employed in the development of algebraic formal methods within Isabelle/HOL (Guttmann et al., 2011).

Isabelle/HOL, Satallax, and LEO-II performed well in recent experiments related to the Flyspeck project (Hales, 2013), in which a formalized proof of the Kepler conjecture is being developed (mainly) in HOL Light; cf. the experiments reported by Kaliszyk and Urban (2012, Table 7).

Most recently, LEO-II, Satallax, and Nitpick were employed to achieve a formalization, mechanization, and automation of Gödel's ontological proof of the existence of God (Benzmüller and Woltzenlogel Paleo, 2013). This work employs a semantic embedding of quantified modal logic in THF0 (Benzmüller and Paulson, 2013). Some previously unknown results were contributed by the provers.

Using the semantic embeddings approach, a wide range of propositional and quantified non-classical logics, including parts of their meta-theory and their combinations, can be automated with THF0 reasoners (cf. Benzmüller (2013); Benzmüller et al. (2012) and Benzmüller (2011)). Automation is thereby competitive, as recent experiments for first-order modal logic show (Benzmüller and Raths, 2013).

THF0 reasoners can also be fruitfully employed for reasoning in expressive ontologies (Benzmüller and Pease, 2012). Furthermore, the heterogeneous toolset HETS (Mossakowski et al., 2007) employs THF0 to integrate the automated higher-order provers Satallax, LEO-II, Nitpick, Refute, and Isabelle/HOL.

8. Conclusion

We have summarized the development of theorem provers for Church's simple theory of types (and elementary type theory) in the 20th century. Given that the model theory and proof theory for ETT, ExTT, and STT is mature, a significant number of interactive and, most recently, automated theorem proving systems have been built for them. Many applications of these systems support Church's original motivation for STT, namely that it could be an elegant, powerful, and mechanized foundations for mathematics. In addition to mathematics, various other application areas (including non-classical logics) are currently being explored.

Acknowledgments. We thank Chad Brown for sharing notes that he has written related to the material in this chapter. Besides the readers of this chapter, we thank Zakaria Chihani, Julian Röder, Leon Weber, and Max Wisnieswki for proofreading the document. The first author has been supported by the German Research Foundation under Heisenberg grant BE2501/9-1 and the second author has been supported by the ERC Advanced Grant ProofCert.

References

Andreoli, J.M., 1992. Logic programming with focusing proofs in linear logic. J. of Logic and Computation 2, 297–347.

Andrews, P., Brown, C., 2006. Tps: A hybrid automatic-interactive system for developing proofs. J. Applied Logic 4, 367–395.

Andrews, P., Cohen, E., 1977. Theorem proving in type theory, in: Proc. of IJCAI-77, 5th International Joint Conference on Artificial Intelligence.

Andrews, P.B., 1965. A Transfinite Type Theory with Type Variables. Studies in Logic and the Foundations of Mathematics, North-Holland Publishing Company.

Andrews, P.B., 1971. Resolution in type theory. Journal of Symbolic Logic 36, 414–432.

Andrews, P.B., 1972a. General models and extensionality. Journal of Symbolic Logic 37, 395–397.

Andrews, P.B., 1972b. General models, descriptions, and choice in type theory. Journal of Symbolic Logic 37, 385–394.

Andrews, P.B., 1973. Letter to Roger Hindley dated January 22, 1973.

Andrews, P.B., 1974. Provability in elementary type theory. Zeitschrift fur Mathematische Logic und Grundlagen der Mathematik 20, 411–418.

Andrews, P.B., 1981. Theorem proving via general matings. J. ACM 28, 193–214.

Andrews, P.B., 1989. On connections and higher order logic. J. of Autom. Reasoning 5, 257–291.

Andrews, P.B., 2001. Classical type theory, in: Robinson, A., Voronkov, A. (Eds.), Handbook of Automated Reasoning. Elsevier Science, Amsterdam. volume 2. chapter 15, pp. 965–1007.

Andrews, P.B., 2002. An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof. Second ed., Kluwer Academic Publishers.

Andrews, P.B., 2009. Church's type theory, in: Zalta, E.N. (Ed.), The Stanford Encyclopedia of Philosophy. spring 2009 ed.. Stanford University.

Andrews, P.B., Bishop, M., Brown, C.E., 2000. TPS: A theorem proving system for type theory, in: McAllester, D. (Ed.), Proceedings of the 17th International Conference on Automated Deduction, Springer, Pittsburgh, USA. pp. 164–169.

Andrews, P.B., Bishop, M., Issar, S., Nesmith, D., Pfenning, F., Xi, H., 1996. TPS: A theorem proving system for classical type theory. J. Autom. Reasoning 16, 321–353.

Andrews, P.B., Longini-Cohen, E., Miller, D., Pfenning, F., 1984. Automating higher order logics. Contemp. Math 29, 169–192.

Arthan, R., 2011. Proofpower website. `http://www.lemma-one.com/ProofPower/index/`.

Autexier, S., Benzmüller, C., Dietrich, D., Siekmann, J., 2010. OMEGA: Resource-adaptive processes in an automated reasoning system, in: Crocker, M.W., Siekmann, J. (Eds.), Resource-Adaptive Cognitive Processes, Springer, Cognitive Technologies. pp. 389–423.

Backes, J., Brown, C.E., 2011. Analytic tableaux for higher-order logic with choice. J. Autom. Reasoning 47, 451–479.

Barendregt, H., 1997. The impact of the lambda calculus in logic and computer science. Bulletin of Symbolic Logic 3, 181–215.

Barendregt, H., Dekkers, W., Statman, R., 2013. Lambda Calculus with Types. Perspectives in Logic, Cambridge University Press.

Beckert, B., Hähnle, R., Schmitt, P.H. (Eds.), 2007. Verification of Object-Oriented Software: The KeY Approach. LNCS 4334, Springer-Verlag.

Beeson, M., 2006. Mathematical induction in Otter-lambda. J. Autom. Reasoning 36, 311–344.

Benl, H., Berger, U., Schwichtenberg, H., Seisenberger, M., Zuber, W., 1998. Proof theory at work: Program development in the minlog system, in: Bibel, W., Schmitt, P. (Eds.), Automated Deduction. Kluwer. volume II.

Benzmüller, C., 1999a. Equality and Extensionality in Automated Higher-Order Theorem Proving. Ph.D. thesis. Saarland University.

Benzmüller, C., 1999b. Extensional higher-order paramodulation and RUE-resolution, in: Ganzinger, H. (Ed.), Proc. of CADE-16, Springer. pp. 399–413.

Benzmüller, C., 2002. Comparing approaches to resolution based higher-order theorem proving. Synthese 133, 203–235.

Benzmüller, C., 2011. Combining and automating classical and non-classical logics in classical higher-order logic. Annals of Mathematics and Artificial Intelligence 62, 103–128.

Benzmüller, C., 2013. Automating quantified conditional logics in HOL, in: Rossi, F. (Ed.), Proc. of IJCAI-23, Beijing, China.

Benzmüller, C., Brown, C., Kohlhase, M., 2004. Higher-order semantics and extensionality. Journal of Symbolic Logic 69, 1027–1088.

Benzmüller, C., Brown, C., Kohlhase, M., 2008a. Cut elimination with xi-functionality, in: Benzmüller, C., Brown, C., Siekmann, J., Statman, R. (Eds.), Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday. College Publications. Studies in Logic, Mathematical Logic and Foundations, pp. 84–100.

Benzmüller, C., Brown, C., Kohlhase, M., 2009. Cut-simulation and impredicativity. Logical Methods in Computer Science 5, 1–21.

Benzmüller, C., Cheikhrouhou, L., Fehrer, D., Fiedler, A., Huang, X., Kerber, M., Kohlhase, M., Konrad, K., Melis, E., Meier, A., Schaarschmidt, W., Siekmann, J., Sorge, V., 1997. OMEGA: Towards a mathematical assistant, in: McCune, W. (Ed.), Proceedings of CADE-14, Springer. pp. 252–255.

Benzmüller, C., Gabbay, D., Genovese, V., Rispoli, D., 2012. Embedding and automating conditional logics in classical higher-order logic. Ann. Math. Artif. Intell. 66, 257–271.

Benzmüller, C., Kohlhase, M., 1998. LEO – a higher-order theorem prover, in: Kirchner, C., Kirchner, H. (Eds.), Proc. of CADE-15, Springer. pp. 139–143.

Benzmüller, C., Paulson, L., 2013. Quantified multimodal logics in simple type theory. Logica Universalis (Special Issue on Multimodal Logics) 7, 7–20.

Benzmüller, C., Pease, A., 2012. Higher-order aspects and context in SUMO. Journal of Web Semantics 12-13, 104–117.

Benzmüller, C., Raths, T., 2013. HOL based first-order modal logic provers, in: McMillan, K., Middeldorp, A., Voronkov, A. (Eds.), Proceedings of LPAR-19, Stellenbosch, South Africa.

Benzmüller, C., Sultana, N., 2013. LEO-II version 1.5, in: Blanchette, J.C., Urban, J. (Eds.), PxTP 2013, EasyChair EPiC Series 14, 2-12. pp. 2–10.

Benzmüller, C., Theiss, F., Paulson, L., Fietzke, A., 2008b. LEO-II - a cooperative automatic theorem prover for higher-order logic (system description), in: Proc. of IJCAR 2008, Springer. pp. 162–170.

Benzmüller, C., Woltzenlogel Paleo, B., 2013. Formalization, Mechanization and Automation of Gödel's Proof of God's Existence. ArXiv e-prints `arXiv:1308.4526`.

Bertot, Y., Casteran, P., 2004. Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer.

Bishop, M., 1999. Mating Search Without Path Enumeration. Ph.D. thesis. Carnegie Mellon University.

Blanchette, J.C., Böhme, S., Paulson, L.C., 2013a. Extending Sledgehammer with SMT solvers. J. Autom. Reasoning 51, 109–128.

Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N., 2013b. Encoding monomorphic and polymorphic types, in: Piterman, N., Smolka, S.A. (Eds.), Proc. of TACAS-19, Springer. pp. 493–507.

Blanchette, J.C., Nipkow, T., 2010. Nitpick: A counterexample generator for higher-order logic based on a relational model finder, in: Kaufmann, M., Paulson, L.C. (Eds.), Proc. of ITP 2010, Springer. pp. 131–146.

Blanchette, J.C., Weber, T., Batty, M., Owens, S., Sarkar, S., 2011. Nitpicking C++ concurrency, in: Schneider-Kamp, P., Hanus, M. (Eds.), Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark, ACM. pp. 113–124.

Bledsoe, W.W., 1979. A maximal method for set variables in automatic theorem-proving, in: Machine Intelligence 9. John Wiley & Sons, pp. 53–100.

Boespflug, M., Carbonneaux, Q., Hermant, O., 2012. The lambda-pi-calculus modulo as a universal proof language, in: Pichardie, D., Weber, T. (Eds.), PxTP 2012, CEUR Workshop Proceedings. pp. 28–43.

Boolos, G., 1987. A curious inference. Journal of Philosophical Logic 16, 1–12.

Brown, C., 2012. Satallax: an automatic higher-order prover. J. Autom. Reasoning , 111–117.

Brown, C.E., 2002. Solving for set variables in higher-order theorem proving, in: Voronkov, A. (Ed.), Proc. of CADE-18, Springer. pp. 408–422.

Brown, C.E., 2004. Set Comprehension in Church's Type Theory. Ph.D. thesis. Department of Mathematical Sciences, Carnegie Mellon University. See also Chad E. Brown, *Automated Reasoning in Higher-Order Logic*, College Publications, 2007.

Brown, C.E., 2005. Reasoning in extensional type theory with equality, in: Nieuwenhuis, R. (Ed.), Proc. of CADE-20, Springer. pp. 23–37.

Brown, C.E., 2013. Reducing higher-order theorem proving to a sequence of sat problems. J. Autom. Reasoning 51, 57–77.

Brown, C.E., Smolka, G., 2010. Analytic tableaux for simple type theory and its first-order fragment. Logical Methods in Computer Science 6.

Bundy, A., van Harmelen, F., Horn, C., Smaill, A., 1990. The Oyster-Clam system, in: Stickel, M.E. (Ed.), 10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings, Springer. pp. 647–648.

Burel, G., 2011a. Efficiently simulating higher-order arithmetic by a first-order theory modulo. Logical Methods in Computer Science 7, 1–31.

Burel, G., 2011b. Experimenting with deduction modulo, in: Bjørner, N., Sofronie-Stokkermans, V. (Eds.), Proc. of CADE-23, Springer. pp. 162–176.

Church, A., 1932. A set of postulates for the foundation of logic. Annals of Mathematics 33, 346–366.

Church, A., 1936. An unsolvable problem of elementary number theory. American Journal of Mathematics 58, 354–363.

Church, A., 1940. A formulation of the simple theory of types. Journal of Symbolic Logic 5, 56–68.

Chwistek, L., 1948. The Limits of Science: Outline of Logic and of the Methodology of the Exact Sciences. London: Routledge and Kegan Paul.

Comon, H., 2001. Inductionless induction, in: Robinson, A., Voronkov, A. (Eds.), Handbook of Automated Reasoning. Elsevier Science. volume I. chapter 14, pp. 913–962.

Constable, R., Allen, S., Bromly, H., Cleaveland, W., Cremer, J., Harper, R., Howe, D., Knoblock, T., Mendler, N., Panangaden, P., Sasaki, J., Smith, S., 1986. Implementing Mathematics with the Nuprl Proof Development System. Prentice-Hall.

Coquand, C., Coquand, T., 1999. Structured type theory, in: Felty, A. (Ed.), Proc. of LMF99: Workshop on Logical Frameworks and Meta-languages.

Cousineau, D., Dowek, G., 2007. Embedding pure type systems in the lambda-pi-calculus modulo, in: Rocca, S.R.D. (Ed.), Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings, Springer. pp. 102–117.

Curry, H., 1942. The inconsistency of certain formal logics. Journal of Symbolic Logic 7, 115–117.

Dixon, L., Fleuriot, J.D., 2003. IsaPlanner: A prototype proof planner in isabelle, in: Baader, F. (Ed.), Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings, Springer. pp. 279–283.

Dowek, G., 1992. Third order matching is decidable, in: 7th Symp. on Logic in Computer Science, IEEE Computer Society Press, Santa Cruz, California. pp. 2–10.

Dowek, G., 1993. A complete proof synthesis method for the cube of type systems. Journal of Logic and Computation 3, 287–315.

Dowek, G., 2001. Higher-order unification and matching, in: Robinson, A., Voronkov, A. (Eds.), Handbook of Automated Reasoning. Elsevier Science, New York. volume II. chapter 16, pp. 1009–1062.

Dowek, G., 2008. Skolemization in simple type theory: the logical and the theoretical points of view, in: Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday. College Publications. number 17 in Studies in Logic, pp. 244–255.

Dowek, G., Hardin, T., Kirchner, C., 2001. HOL-$\lambda\sigma$ an intentional first-order expression of higher-order logic. Mathematical Structures in Computer Science 11, 1–25.

Dowek, G., Hardin, T., Kirchner, C., 2003. Theorem proving modulo. J. Autom. Reasoning 31, 33–72.

Enderton, H.B., 1972. A Mathematical Introduction to Logic. Academic Press.

Enderton, H.B., 2012. Second-order and higher-order logic, in: Zalta, E.N. (Ed.), The Stanford Encyclopedia of Philosophy. fall 2012 ed.. Stanford University.

Farmer, W.M., 1990. A partial functions version of church's simple theory of types. J. Symb. Log. 55, 1269–1291.

Farmer, W.M., 1993. IMPS: An interactive mathematical proof system. J. Autom. Reasoning 11, 213–248.

Farmer, W.M., 2008. The seven virtues of simple type theory. J. Applied Logic 6, 267–286.

Farmer, W.M., Guttman, J.D., Thayer, F.J., 1992. Little theories, in: Kapur, D. (Ed.), Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings, Springer. pp. 567–581.

Felty, A., 2000. The calculus of constructions as a framework for proof search with set variable instantiation. Theoretical Computer Science 232, 187–229.

Frege, G., 1879. Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens. Halle. Translated in van Heijenoort (1967).

Gacek, A., Miller, D., Nadathur, G., 2012. A two-level logic approach to reasoning about computations. J. Autom. Reasoning 49, 241–273.

Gentzen, G., 1969a. Investigations into logical deduction, in: Szabo, M.E. (Ed.), The Collected Papers of Gerhard Gentzen. North-Holland, Amsterdam, pp. 68–131. Translation of articles that appeared in 1934-35.

Gentzen, G., 1969b. New version of the consistency proof for elementary number theory, in: Szabo, M.E. (Ed.), Collected Papers of Gerhard Gentzen. North-Holland, Amsterdam, pp. 252–286. Originally published 1938.

Girard, J.Y., 1971. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types, in: Fenstad, J.E. (Ed.), 2nd Scandinavian Logic Symposium. North-Holland, Amsterdam, pp. 63–92.

Girard, J.Y., 1986. The system F of variable types: Fifteen years later. Theoretical Computer Science 45, 159–192.

Gödel, K., 1929. Über die Vollständigkeit des Logikkalküls. Ph.D. thesis. Universität Wien.

Gödel, K., 1930a. Die vollständigkeit der axiome des logischen funktionenkalküls. Monatshefte für Mathematik 37, 349360.

Gödel, K., 1930b. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. Monatshefte für Mathematik und Physik 37, 349–360.

Gödel, K., 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte der Mathematischen Physik 38, 173–198. English Version in van Heijenoort (1967).

Gödel, K., 1936. Über die Länge von Beweisen, in: Ergebnisse eines Mathematischen Kolloquiums, pp. 23–24. English translation "On the length of proofs" in Kurt Gödel: Collected Works, Volume 1, pages 396-399, Oxford University Press, 1986.

Goldfarb, W., 1981. The undecidability of the second-order unification problem. Theoretical Computer Science 13, 225–230.

Gordon, M., Melham, T., 1993. Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic. Cambridge University Press.

Gordon, M.J.C., Milner, R., Wadsworth, C.P., 1979. Edinburgh LCF. volume 78 of *LNCS*. Springer.

Gould, W.E., 1966. A Matching Procedure for ω-Order Logic. Technical Report Scientific Report No. 4. A F C R L.

Guard, J.R., 1964. Automated logic for semi-automated mathematics, in: Scientific Report No 1. A F C R L, pp. 64–411.

Guttmann, W., Struth, G., Weber, T., 2011. Automating algebraic methods in isabelle, in: Qin, S., Qiu, Z. (Eds.), Proc. of ICFEM 2011, Springer. pp. 617–632.

Hales, T., 2013. Mathematics in the Age of the Turing Machine. ArXiv e-prints `arXiv:1302.2898`.

Harrison, J., 2009. HOL Light: An overview, in: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (Eds.), Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings, Springer. pp. 60–66.

van Heijenoort, J., 1967. From Frege to Gödel: A Source Book in Mathematics, 1879-1931. Source books in the history of the sciences series. 3rd printing, 1997 ed., Harvard Univ. Press, Cambridge, MA.

Henkin, L., 1950. Completeness in the theory of types. Journal of Symbolic Logic 15, 81–91.

Henkin, L., 1963. A theory of propositional types. Fundamatae Mathematicae , 323–344.

Henschen, L.J., 1972. N-sorted logic for automatic theorem-proving in higher-order logic, in: Proceedings of the ACM Annual Conference - Volume 1, ACM, New York, NY, USA. pp. 71–81.

Hermant, O., Lipton, J., 2010. Completeness and cut-elimination in the intuitionistic theory of types - part 2. J. Logic and Computation 20, 597–602.

Hilbert, D., 1899. Grundlagen der geometrie, in: Festschrift zur Feier der Enthllung des Gauss-Weber-Denkmals in Göttingen. B.G. Teubner, Leipzig, pp. 1–92.

Holmes, M.R., Alves-Foss, J., 2001. The Watson theorem prover. J. Autom. Reasoning 26, 357–408.

Huet, G., 1973a. The undecidability of unification in third order logic. Information and Control 22, 257–267.

Huet, G., 1975. A unification algorithm for typed λ-calculus. Theoretical Computer Science 1, 27–57.

Huet, G., Lang, B., 1978. Proving and applying program transformations expressed with second-order patterns. Acta Informatica 11, 31–55.

Huet, G.P., 1972. Constrained Resolution: A Complete Method for Higher Order Logic. Ph.D. thesis. Case Western Reserve University.

Huet, G.P., 1973b. A mechanization of type theory, in: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, pp. 139–146.

Hurd, J., 2003. First-order proof tactics in higher-order logic theorem provers, in: Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports, pp. 56–68.

Jensen, D.C., Pietrzykowski, T., 1976. Mechanizing *omega*-order type theory through unification. Theor. Comput. Sci. 3, 123–171.

Kaliszyk, C., Urban, J., 2012. Learning-assisted automated reasoning with flyspeck. CoRR abs/1211.7012.

Kaufmann, M., Moore, J.S., 1997. An industrial strength theorem prover for a logic based on Common Lisp. IEEE Trans. Software Eng. 23, 203–213.

Kerber, M., 1991. How to prove higher order theorems in first order logic, in: Mylopoulos, J., Reiter, R. (Eds.), Proc. of IJCAI-12, Morgan Kaufmann. pp. 137–142.

Kerber, M., 1994. On the translation of higher-order problems into first-order logic, in: Proc. of ECAI, pp. 145–149.

Kleene, S., Rosser, J., 1935. The inconsistency of certain formal logics. Annals of Mathematics 36, 630–636.

Kohlhase, M., 1994. A Mechanization of Sorted Higher-Order Logic Based on the Resolution Principle. Ph.D. thesis. Saarland University.

Leivant, D., 1994. Higher-order logic, in: Gabbay, D.M., Hogger, C.J., Robinson, J.A. (Eds.), Handbook of Logic in Artificial Intelligence and Logic Programming. Oxford University Press. volume 2, pp. 229–321.

Liang, C., Miller, D., 2009. Focusing and polarization in linear, intuitionistic, and classical logics. Theoretical Computer Science 410, 4747–4768.

Liang, C., Nadathur, G., Qi, X., 2005. Choices in representing and reduction strategies for lambda terms in intensional contexts. Journal of Automated Reasoning 33, 89–132.

Lindblad, F., 2013. agsyHOL website. `https://github.com/frelindb/agsyHOL`.

Lucchesi, C.L., 1972. The Undecidability of Unification for Third Order Languages. Technical Report Report CSRR 2059. Dept of Applied Analysis and Computer Science, University of Waterloo.

Martin-Löf, P., 1982. Constructive mathematics and computer programming, in: Sixth International Congress for Logic, Methodology, and Philosophy of Science, North-Holland, Amsterdam. pp. 153–175.

McDowell, R., Miller, D., 2002. Reasoning with higher-order abstract syntax in a logical framework. ACM Trans. Comput. Log. 3, 80–136.

Meng, J., Paulson, L.C., 2008. Translating higher-order clauses to first-order clauses. J. Autom. Reasoning 40, 35–60.

Miller, D., 1983. Proofs in Higher-Order Logic. Ph.D. thesis. Carnegie-Mellon University.

Miller, D., 1987. A compact representation of proofs. Studia Logica 46, 347–370.

Miller, D., 1991. A logic programming language with lambda-abstraction, function variables, and simple unification. Journal of Logic and Computation 4, 497–536.

Miller, D., 1992. Unification under a mixed prefix. Journal of Symbolic Computation 14, 321–358.

Miller, D., 2011. A proposal for broad spectrum proof certificates, in: Jouannaud, J.P., Shao, Z. (Eds.), CPP: First International Conference on Certified Programs and Proofs, pp. 54–69.

Miller, D., Nadathur, G., 2012. Programming with Higher-Order Logic. Cambridge University Press.

Miller, D.A., Cohen, E.L., Andrews, P.B., 1982. A look at TPS, in: Loveland, D.W. (Ed.), Sixth Conference on Automated Deduction, Springer, New York. pp. 50–69.

Mints, G., 1999. Cut-elimination for simple type theory with an axiom of choice. J. Symb. Log. 64, 479–485.

Mossakowski, T., Maeder, C., Lüttich, K., 2007. The heterogeneous tool set, Hets, in: Proceedings of TACAS 2007, Springer. pp. 519–522.

Muskens, R., 2007. Intensional models for the theory of types. J. Symb. Log. 72, 98–118.

Nadathur, G., Linnell, N., 2005. Practical higher-order pattern unification with on-the-fly raising, in: ICLP 2005: 21st International Logic Programming Conference, Springer, Sitges, Spain. pp. 371–386.

Nadathur, G., Miller, D., 1988. An Overview of λProlog, in: Fifth International Logic Programming Conference, MIT Press, Seattle. pp. 810–827.

Nederpelt, R.P., Geuvers, J.H., Vrijer, R.C.D. (Eds.), 1994. Selected Papers on Automath. volume 133 of *Studies in Logic and The Foundations of Mathematics*. North Holland.

Nipkow, T., 1989. Equational reasoning in Isabelle. Sci. Comput. Program. 12, 123–149.

Nipkow, T., Paulson, L., Wenzel, M., 2002. Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Number 2283 in LNCS, Springer.

Owre, S., Rushby, J., Shankar, N., 1992. PVS: A Prototype Verification System, in: D., K. (Ed.), Proceedings of the 11th International Conference on Automated Deduction, Springer. pp. 748–752.

Padovani, V., 2000. Decidability of fourth-order matching. Mathematical Structures in Computer Science 10, 361–372.

Parikh, R.J., 1973. Some results on the length of proofs. Transactions of the ACM 177, 29–36.

Paulson, L.C., 1989. The foundation of a generic theorem prover. J. Autom. Reasoning 5, 363–397.

Paulson, L.C., 1994. Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow). volume 828 of *LNCS*. Springer.

Paulson, L.C., 1999. A generic tableau prover and its integration with isabelle. Journal of Universal Computer Science 5, 51–60.

Peano, G., 1889. Arithmetices principia, nova methodo exposita. Fratres Bocca, Turin.

Pfenning, F., 1987. Proof Transformations in Higher-Order Logic. Ph.D. thesis. Carnegie Mellon University. 156 pp.

Pfenning, F., 1994. Elf: A meta-language for deductive systems (system descrition), in: Bundy, A. (Ed.), Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings, Springer. pp. 811–815.

Pfenning, F., Schürmann, C., 1999. System description: Twelf - a meta-logical framework for deductive systems, in: Ganzinger, H. (Ed.), Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings, Springer. pp. 202–206.

Pientka, B., Dunfield, J., 2010. Beluga: A framework for programming and reasoning with deductive systems (system description), in: Giesl, J., Hähnle, R. (Eds.), Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings, Springer. pp. 15–21.

Pietrzykowski, T., 1973. A complete mechanization of second-order type theory. J. ACM 20, 333–364.

Pietrzykowski, T., Jensen, D.C., 1972. A complete mechanization of ω-order type theory, in: ACM '72: Proceedings of the ACM annual conference, ACM Press, New York, NY, USA. pp. 82–92.

Pollack, R., 1994. The Theory of LEGO. Ph.D. thesis. University of Ediburgh.

Prawitz, D., 1968. Hauptsatz for higher order logic. Journal of Symbolic Logic 33, 452–457.

Quine, W.V.O., 1940. Mathematical Logic. Harvard University Press, Boston, MA.

Ramsey, F.P., 1926. The foundations of mathematics, in: Proceedings of the London Mathematical Society, pp. 338–384.

Reynolds, J.C., 1974. Towards a theory of type structure, in: Colloque sur la Programmation, Paris, France, Springer, New York. pp. 408–425.

Richardson, J., Smaill, A., Green, I., 1998. System description: Proof planning in higher-order logic with lambda-Clam, in: Kirchner, C., Kirchner, H. (Eds.), Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings, Springer. pp. 129–133.

Robinson, J.A., 1969. Mechanizing higher-order logic, in: Machine Intelligence 4. Edinburgh University Press, pp. 151–170.

Robinson, J.A., 1970. A note on mechanizing higher order logic, in: Machine Intelligence 5. Edinburgh University Press, pp. 121–135.

Russell, B., 1902. Letter to Frege. Translated in van Heijenoort (1967).

Russell, B., 1903. The principles of mathematics. Cambridge University Press, Cambridge, England.

Russell, B., 1908. Mathematical logic as based on the theory of types. American Journal of Mathematics 30, 222–262.

Schulz, S., 2002. E – a brainiac theorem prover. AI Communications 15, 111–126.

Schütte, K., 1960. Semantical and syntactical properties of simple type theory. Journal of Symbolic Logic 25, 305–326.

Shapiro, S., 1985. Second-order languages and mathematical practice. Journal of Symbolic Logic 50, 714–742.

Smullyan, R.M., 1963. A unifying principle for quantification theory. Proc. Nat. Acad Sciences 49, 828–832.

Snyder, W., Gallier, J.H., 1989. Higher order unification revisited: Complete sets of transformations. Journal of Symbolic Computation 8, 101–140.

Spenger, C., Dams, M., 2003. On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ-calculus, in: Gordon, A. (Ed.), FOSSACS'03, Springer. pp. 425–440.

Stirling, C., 2009. Decidability of higher-order matching. Logical Methods in Computer Science 5, 1–52.

Sutcliffe, G., 2009. The TPTP problem library and associated infrastructure. Journal of Automated Reasoning 43, 337–362.

Sutcliffe, G., Benzmüller, C., 2010. Automated reasoning in higher-order logic using the TPTP THF infrastructure. Journal of Formalized Reasoning 3, 1–27.

Tait, W.W., 1966. A nonconstructive proof of Gentzen's Hauptsatz for second order predicate logic. Bulletin of the American Mathematical Society 72, 980983.

Takahashi, M., 1967. A proof of cut-elimination theorem in simple type theory. Journal of the Mathematical Society of Japan 19, 399–410.

Takeuti, G., 1953. On a generalized logic calculus. Japanese Journal of Mathematics 23, 39–96. Errata: ibid, vol. 24 (1954), 149–156.

Takeuti, G., 1960. An example on the fundamental conjecture of GLC. Journal of the Mathematical Society of Japan 12, 238–242.

Takeuti, G., 1975. Proof Theory. volume 81 of *Studies in Logic and the Foundations of Mathematics*. Elsevier.

Westerståhl, D., 2011. Generalized quantifiers, in: Zalta, E.N. (Ed.), The Stanford Encyclopedia of Philosophy. summer 2011 ed.

Whitehead, A.N., Russell, B., 1910, 1912, 1913. Principia Mathematica, 3 vols. Cambridge: Cambridge University Press. Second edition, 1925 (Vol. 1), 1927 (Vols 2, 3). Abridged as Principia Mathematica to *56, Cambridge: Cambridge University Press, 1962.

Wirth, C.P., 2004. Descente infinie + deduction. Logic Journal of the IGPL 12, 1–96.