UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Modulárne programovanie a verifikácia v druhorádovej aritmetike

Dizertačná práca

Vedný odbor: 25-11-9 Aplikovaná informatika
Školiace pracovisko: Katedra aplikovanej informatiky
Školiteľ: doc. RNDr. Pavol J. Voda, PhD.

Bratislava 2010

Mgr. Ján Kľuka

Abstrakt

V predloženej dizertačnej práci sa zaoberáme niektorými aspektmi deklaratívneho programovania a verifikácie deklaratívnych programov. Vychádzame z existujúceho deklaratívneho programovacieho jazyka a špecifikačného a verifikačného systému CL (Clausal Language).

Súčasná verzia CL je založená na Peanovej aritmetike. Programovací jazyk CL je expresívny a pohodlný. Kvôli prvorádovému logickému základu mu však chýba dôležitá vlastnosť pre podporu modulárneho programovania, a to zapúzdrenie vnútorných komponentov modulov.

Naším prvým výsledkom je návrh na zmenu logickej teórie, na ktorej je CL založené, z Peanovej aritmetiky na istú slabú druhorádovú teóriu aritmetiky. Vysvetľujeme, ako táto teória podporuje modularitu, a analyzujeme jej silu. Navrhujeme tiež nový formálny systém pre druhorádovú logiku, kalkul ohodnocovacích stromov. Jeho jediným odvodzovacím pravidlom je rez (cut) z gentzenovských systémov.

Programy sa v CL verifikujú pomocou dokazovacieho asistenta. Dokazovací asistent umožňuje používateľovi konštruovať formálne dôkazy. Jeho automatická časť dokáže dokončiť niektoré dôkazy pomocou rozhodovacích procedúr pre vlastnosti operácií zabudovaných do jazyka CL a symbolickým vyhodnocovaním CL programov. Automatická časť však nedokáže využiť lemy dokázané používateľom.

Naším druhým výsledkom je nový algoritmus pre automatickú časť dokazovacieho asistenta, schopný používať lemy, ktoré majú tvar všeobecne kvantifikovaných klauzúl. Nový algoritmus vychádza z algoritmu pre kongruenčný uzáver, ktorý rozhoduje logické vyplývanie rovností termov bez premenných z množín takýchto rovností. Nový algoritmus porovnáva rovnosti vo všeobecne kvantifikovaných klauzulách s rovnosťami odvodenými kongruenčným uzáverom. Algoritmus je neúplný, dokladáme však jeho užitočnosť pri praktickom dokazovaní vlastností programov.

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS, AND INFORMATICS

Modular Programming and Verification in Second-Order Arithmetic

PhD. thesis

Field of study: 25-11-9 Applied informatics
Advising department: Department of Applied Informatics
Advisor: doc. RNDr. Paul J. Voda, PhD.

Bratislava 2010

Mgr. Ján Kľuka

Abstract

The present thesis is concerned with some aspects of declarative programming and verification of declarative programs. We build on the existing declarative programming language, and specification and verification system CL (Clausal Language).

The current version of CL is based on Peano arithmetic. The CL programming language is expressive and comfortable. However, due to its first-order logical basis, it lacks an important feature for support of modular programming, namely encapsulation of internal components of modules.

Our first contribution is a proposal to change the underlying theory of CL from Peano arithmetic to a particular weak second-order theory of arithmetic. We explain how this theory supports modularity, and analyze its power. We also propose a new formal system for second-order logic, the valuation tree calculus. The only inference rule of the calculus is the cut from Gentzen's calculi.

Programs are verified in CL with the help of a proof assistant. The proof assistant enables the user to construct formal proofs. Its automatic part can complete some proofs using decision procedures for properties of built-in CL operations, and using symbolic evaluation of CL programs. However, the automatic part is unable to use lemmas proved by the user.

Our second contribution is a new algorithm for the automatic part of the proof assistant which can use lemmas which have the form of universally quantified clauses. The new algorithm is based on the congruence closure algorithm which decides logical consequence of equalities of variable-free terms from sets of such equalities. The new algorithm matches equalities in universally quantified lemmas with equalities derived by the congruence closure. The algorithm is incomplete, but we present evidence supporting its practical usefulness.

Preface

Formal verification of programs considerably increases confidence in correctness of software in mission-critical applications. Verification can also be beneficial less directly by improving the understanding of the problem being solved by the verified program, and the understanding of the program itself.

Verification requires existence of a formal specification written in a formal logical language. In order to verify a program written in an imperative programming language such as C or Java, its semantics must be first expressed in the same logical language as the formal specification.

The necessity to express the meaning of a program to the language of the specification is eliminated in declarative programming languages. While imperative programs are primarily viewed as sequences of instructions for changing state, declarative programs are mathematical objects such as functions or relations. Declarative programs can be easily translated to definitions in a suitable kind of logic. In the logic, specifications can be written as formulas and proved as theorems.

In late 1996, Paul J. Voda and Ján Komara have started a project of a declarative programming language and verification system called CL (Clausal Language). The goal of the CL project has been to design an expressive and comfortable programming language with as simple logical foundation as possible. Since the inception of the project, the foundation has been Peano arithmetic, the first-order theory of natural numbers. CL is implemented as an interactive environment in which programs can be specified, implemented, run, and formally verified using a proof assistant. This system is used at Comenius University in teaching of undergraduate courses in declarative programming, and in specification and verification of programs.

The present thesis contributes to two aspects of the CL project. First, we address the shortcomings of the CL language in the area of modular programming. Second, we propose an algorithm which will enable more automation in the CL proof assistant, while permitting explanation of automatic proofs. This work will lead to a new implementation of the CL system.

This thesis was written while the author studied at the Department of Applied Informatics (and the former Institute of Informatics) of Faculty of Mathematics, Physics, and Informatics of Comenius University in Bratislava. The author would like to thank his advisor, Paul J. Voda, for guidance, encouragement, patience, comments, and numerous "wild" ideas some of which are "tamed" in this thesis. The practical evaluation of a part of this thesis would be impossible without Ján Komara, who implemented the current CL proof assistant, and who created and maintains a large library of verified programs used in the specification and verification courses. The author would also like to thank Ján Šefránek, the former head of Institute of

Informatics, Ján Rybár, the current head of the Department of Applied Informatics, and other colleagues for the friendly and liberal atmosphere at the Department. Last, but not least the author would like to thank his family and friends for their support and patience.

Contents

1 Introduction

The present thesis proposes a logical basis for formally verified modular programming. We build upon the existing system CL (Clausal Language). CL is used in teaching of declarative programming and program verification at Comenius University since 1997. The work described in the thesis will eventually lead to a new implementation of CL.

1.1 The Current System CL

1.1.1 Programming language and logic. CL is a declarative programming language build on a simple and purely logical basis of Peano arithmetic (PA). In the more common imperative paradigm of programming, programs are sequences of instructions modifying the state of a computer's memory and input-output devices. On the other hand, declarative programs are mathematical objects, usually functions (in functional programming languages such as ML or Haskell) or relations (in logical languages such as Prolog). A purely declarative program yields an output value for a given input without any *side effects*, i.e., without any changes of state beyond those described by the input and output values. In declarative programming, emphasis is put on *what* a program computes, rather than on *how* the result is obtained. This emphasis and the lack of side effects make declarative programs easier to verify against a given specification.

Programs in CL are primitive recursive functions on numbers defined by formulas of PA. Development of CL from PA is described in the textbook [Vod01a]. Since PA deals with natural numbers only, data structures are arithmetized, i.e., encoded into numbers. Towards this end, PA is extended with a pairing function, a bijection of $\mathbb{N}^2$ onto $\mathbb{N} \setminus \{0\}$. The usual data structures of functional programming languages — tuples, lists, and recursive tree-like structures — are then encoded as pairs.

The basic logical tool for extending theories with new function symbols are contextual definitions of the form $f(x) = y \leftrightarrow A[x, y]$. Such definitions are conservative, i.e., they do not essentially change logical consequences of the theory. But they do not allow recursion, and can define non-computable functions. Definitions by primitive recursion, such as

$$\begin{aligned} &\mathrm{F}(0) = 1 \\ &\mathrm{F}(x+1) = (x+1) \cdot \mathrm{F}(x) \end{aligned}$$

guarantee computability and are reducible to contextual definitions. However, the standard schema of primitive recursion is inconvenient for defining functions operating on arithmetized data structures.

Therefore, CL uses a very general schema of recursive clausal definitions with measure. A definition in this schema is a set of clauses similar to a Prolog program, but with additional constraints. Clauses must be mutually exclusive, since otherwise they could "define" a function to have different values in overlapping cases. Mutual exclusivity is ensured by syntactic constraints that can be machine-checked.

Recursion in clausal definitions is restricted by a semantic condition: There must be an independent measure function, in which the recursive arguments are less than the original arguments. For instance, primitive recursion is a simple special case with identity as measure ($\mathrm{id}(x) < \mathrm{id}(x{+}1)$). Clausal recursive definitions with measure are still reducible to contextual definitions, and define only primitive recursive functions. Properties of clausally defined programs can be expressed by formulas of PA (e.g., $\forall x(\mathrm{F}(x) > 0)$ for F defined above) and formally proved.

1.1.2 Implementation. An evolving implementation of CL has been used by hundreds of students of several courses at Comenius University since 1997. The implementation consists of (i) a compiler of clausal definitions, (ii) a byte-code interpreter allowing computation of defined functions, and (iii) a proof assistant. These components are integrated into a program and proof development environment. Its user interface is build in XHTML and MathML ([CIMP03]), and accessed via a web browser. The interface allows users to interactively edit CL modules, which are sequences consisting of function definitions and theorems with formal proofs. Users can also invoke the interpreter to evaluate functions.

The CL language was designed by Paul J. Voda and Ján Komara. Paul J. Voda implemented the compiler and byte-code interpreter, and Ján Komara implemented the proof assistant. The present author contributed the XHTML+MathML user interface. The current version of CL is available from download at [CL].

1.2 Selected Problems in the Current Version of CL

The present thesis is concerned with the logical basis of the planned new version of CL, and with the proof assistant component of the implementation. In the following paragraphs, we will identify the problems in the current version of CL which are addressed in this thesis.

1.2.1 Extensions of PA and modularity. Only very simple programs can be written in CL as a single function definition. Usually a series of definitions is needed, starting with auxiliary functions and ending with the main function of the program. For verification of the program, it is necessary to follow each definition with proofs of properties of the defined function.

Such a series of definitions is logically a sequence of *extensions by definitions* of the theory PA (cf. [Sho67, §4.6]). For example, we extend PA by a definition of a function f, thus obtaining a theory PA_f, in which we prove a property A_f of f.

Then, we extend PA_f by a definition of a function g, obtaining PA_{fg}, in which we prove a property A_g of g; etc.

Often, the existence of a function g with the property A_g is the only information needed for further development and verification of the program. The function f and its property A_f are needed to define g and prove A_g, but they are not used anywhere else in the program. In the programming terminology, the functions f and g along with their properties constitute a *module*. The property A_g is the *interface* of the module. The function f, its property A_f, and the actual definition of g are the *implementation* of the module.

In modular programming languages, only the interface of a module is exposed, and the implementation is hidden (*encapsulated*) from the rest of the program. Modular development is a basic good practice in programming. It allows to structure the program on a higher level to components communicating only through relatively simple and stable interfaces. Implementation of a module can change (e.g., to use a more efficient internal data structure) without affecting the rest of the program. Modular languages also allow implementation components of different modules to use identical names. Such a feature is logically uninteresting, but practically important, since complex programs consist of thousands of functions.

Let us return to the above example. Since, from the logical point of view, any function h using the function g needs to be defined in the extension PA_{fg}, the implementation of g cannot be hidden. The current version of CL thus does not support encapsulation, a crucial feature enabling modular programming. We will discuss the modularity issues in more detail in Sect. 2.1.

1.2.2 Proof assistants. *Proof assistants* (also called interactive theorem provers or proof checkers) are tools for construction of formal proofs. They require a human user to provide the crucial steps of the proof, such as a choice of the induction formula, or terms to substitute into a lemma. Proof assistants check admissibility of human-provided steps, and attempt to fill in the rest.

On the other hand, *automated theorem provers* try to prove given formulas without human intervention. Full automation is often desired, and automated provers are successfully applied to verification of hardware, communication protocols, distributed systems, or even simpler safety properties of programs. However, systems verified in these applications either have finitely many states, or they have potentially infinitely many states which can be abstracted to finitely many classes.

Verification of programs against detailed formal specifications requires considering infinitely many states with the help of induction. Choosing a suitable induction hypothesis often requires ingenious abstraction, which calls for human intervention, and therefore for the use of a proof assistant.

Several proof assistants are in active development and use. Some of the most prominent are Isabelle [NPW02, Wen02], NuPRL [CAB+86, Kre02], Coq [CDT06], PVS [SORS01]. These proof assistants work in various higher-order logics, and allow proving properties of functional programs.

Proof assistants typically have two parts, loosely depending on each other:

(i) A *proof-building* part enables the user to construct proofs by applying rules of a human-oriented formal system. Common choices are Gentzen's natural deduction or sequent calculus, or Beth's and Smullyan's semantic tableaux.

(ii) An *automatic* part of the assistant frees the user from manual application of simple rules (such as replacement of equals by equals) and from proving simple properties of, e.g., lists or linear arithmetic. The core of this part usually consists of a congruence closure algorithm to decide equalities of variable-free terms, combined with decision procedures for simple theories. Ideally, the automatic part should explain and certify its deductions by generating a proof in the formal system use by the proof-building part. However, automated deduction algorithms do not directly correspond to rules of human-oriented formal systems. Thus explanation of automated steps is rarely present.

1.2.3 The current CL proof assistant. The interactive proof-building part of the existing CL proof assistant uses a variation of semantic tableaux with signed formulas. Besides standard tableau rules, there are several derived rules, notably induction.

After the user applies a tableau rule, the automatic part of the assistant is executed. It uses Shostak's approach to congruence closure and combination of decision procedures (see, e.g., [RS01]). The main principle of this approach is rewriting of terms to a canonical form. The automatic part either completes the proof, or canonizes current assumptions and goals. The resulting canonical forms are presented to the user, who can choose the next tableau rule.

Canonical forms are often expected and helpful. They give the user a sense of progress, and may indicate which information is missing to complete the proof. On the other hand, canonization in many cases obscures the original problem instead of simplifying it. In the example from Par. 1.1.1, the canonical form of $\mathrm{F}(n+1)$ is $n{\cdot}\mathrm{F}(n) + \mathrm{F}(n)$. This form is not useful when we are proving that $n+1$ divides $\mathrm{F}(n+1)$, or when we need to infer $\mathrm{F}(n+1) > 0$ from a lemma $\forall x(\mathrm{F}(x) > 0)$. Overly complex canonical forms are especially unfortunate in classroom situations. Inexperienced users easily get confused and loose track of the original goal. Since canonization plays a prominent role in the current CL implementation, it cannot be easily suppressed.

The automatic part of the CL proof assistant does not currently explain why it finished a proof. Therefore, the proof cannot be independently checked in order to catch possible errors in the complex inner workings of the automatic part. Another undesired effect is that students tend to view the assistant as a black box, and often try to make it finish a proof by random trial and error.

The proof assistant has a built-in knowledge of PA (various non-trivial properties of addition, multiplication, and ordering of natural numbers), and can also decide properties of arithmetized data structures (pairs, lists). It also uses clausal definitions to canonize terms containing user-defined functions. However, the assistant cannot automatically use already proved properties of these functions (lemmas).

To sum up, we have identified three problems in the current CL proof assistant, which we would like to address in the new implementation:

(i) excessive canonization,
(ii) lack of independently checkable explanation, and
(iii) inability to use lemmas.

1.3 Our Contribution

We propose two improvements on the current design which address the problems identified in the previous section. The first improvement is an extension of the PA to the second order, which will allow modular programming in CL. We accompany the second order extension of PA with a new formal system. The formal system is to be used in the proof-building part of the proof assistant. It will also replace the current mechanism of program development by successive extensions by definitions. The second improvement is a algorithm for the automatic part of the proof assistant, which is capable of using user-proved lemmas.

1.3.1 A second-order theory and formal system for modular programming. In Chapter 2, we describe a new formal system for second-order logic and a weak second-order extension CL_2 of PA.

We plan to use the theory CL_2 as the logical basis of the new CL implementation, instead of PA. Second-order logic allows an elegant logical explanation of modular programming. Recall the example from Par. 1.2.1. The module in the example can be logically viewed as a second-order theorem $\exists g\, A_g$ taken together with its proof. The formula $\exists g\, A_g$ is the interface of the module, and the proof of $\exists g\, A_g$ is the implementation of the module. Within the proof, we can define the auxiliary function f and the exposed function g. A more complex example will be presented in Sect. 2.8. Moreover, we will show in Par. 2.7.7 that CL_2 is equivalent to RCA_0, a well known weak subsystem of full second-order arithmetic Z_2. RCA_0 plays an important role in the study of foundations of mathematics ([Sim99]).

Additionally to the theory CL_2, we will present in Chapter 2 the *valuation tree calculus*, a new formal system for second-order logic. This calculus is based on a first-order version presented in [KV09]. The planned new implementation of CL will use the valuation tree calculus in the proof-building part of its proof assistant instead of current version's tableau calculus. The valuation tree calculus has several features which we consider beneficial in our context:

(a) Valuation tree calculus has one rule (the cut from Gentzen's sequent calculus) and many axioms, much like the Hilbert system with modus ponens. The *cut* on a formula A is a simple rule which splits a proof into two branches: one in which A is assumed to hold, and the other in which A has to be proved.

Theoretical analysis of a formal system with one rule is simpler compared to many-rule Gentzen or tableaux systems. From the implementation point of view, a single rule simplifies the internal representation of formal proofs.

(b) Proofs in the valuation tree calculus can be constructed in the human-oriented style of Gentzen's sequent calculus or Smullyan's semantic tableaux.

The two common single-rule systems, namely the Hilbert system and resolution, do not enjoy this property: Construction of actual formal proofs is quite difficult and counterintuitive in the Hilbert system without the help of the deduction theorem. Resolution was invented specifically as a machine-oriented formal system ([Rob65]), and is widely regarded as unfit for human use.

The relationship of the valuation tree calculus, sequent calculus, and tableaux will be discussed in Sect. 2.4.

(c) CL currently has two modes:
  (i) a theory extension mode, in which PA is extended by clausal definitions of functions, and their properties are asserted;
  (ii) a proof-building mode, in which the asserted properties are proved using the proof assistant.

The valuation tree calculus allows for introduction of new function and predicate symbols within a proof. Due to this feature, we can incorporate the theory extension mode into the proof-building mode. The unification of proofs and theory extensions will be discussed in Sect. 2.5.

1.3.2 Congruence closure with instantiation of clauses. In Chapters 3 through 5, we theoretically develop an improvement on the automatic part of the proof assistant.

We start with the congruence closure algorithm in Chapter 3. This algorithm decides whether an equality of variable-free terms is implied by a set of such equalities. Several versions of the algorithm were described in literature in terms of directed acyclic graphs or term rewriting ([NO80, DST80, NO03]). We describe the algorithm as a transformation of trees. We believe this view is more intuitive than the existing descriptions.

In Chapter 4, we first show that the congruence closure can also decide equality of terms implied by a set of quantifier- and variable-free Horn clauses. We then discuss an extension to quantifier- and variable-free general clauses, which we have presented in [KIu05].

Finally, in Chapter 5, we develop an approach to decision of equality of variable-free terms from sets of generally quantified clauses. We simplify this undecidable problem as follows: We assume there is a fixed set of variable-free terms, a domain D. We weaken all clauses by adding a requirement that all terms are equal to a member of D. We call such clauses *domain-bounded*. Equality of variable-free terms implied by domain-bounded clauses then becomes decidable.

We develop a special-purpose *BC proof system* for this simplified problem, and show it is complete. The BC proof system can be briefly described as follows: The set of domain-bounded clauses is split into a set of equalities of variable-free terms, and a set of proper generally quantified clauses. A congruence closure of the equalities is extended to all terms in D. Equalities in antecedents of proper clauses are matched against those in the congruence closure. Matching produces new clauses, eventually finding substitutions for variables, and deriving new equalities of variable-free terms.

For example, assume that the input set Π consists of equalities

$$f(c, f(u, v)) = f(e, d) \qquad \text{and} \qquad c = e,$$

and the clause

$$\forall x \forall y \forall z (f(x, y) = f(x, z) \rightarrow y = z) \tag{1}$$

of the left cancellation law for f. We will show how $f(u, v) = d$ is derived from Π.

The congruence closure of the input equalities produces two non-trivial equivalence classes:

$$C_1 = \{c, e\} \qquad C_2 = \{f(c, f(u, v)), f(e, d)\}.$$

Now consider the equality in the antecedent of the cancellation law (1). The top-level function symbol on both sides is f. The equivalence class C_2 contains terms with f at the top level. Therefore, we can match those terms with the antecedent of (1) and derive a clause

$$\forall x \forall y \forall z (c = x \wedge f(u, v) = y \wedge e = x \wedge d = z \rightarrow y = z). \tag{2}$$

The clause (2) is a logical consequence of the input set Π. By substitution, we can derive the following logical consequence of (2)

$$c = e \rightarrow f(u, v) = d, \tag{3}$$

which, of course, is a logical consequence of Π. Since $c = e$ is also a logical consequence of Π, we derive $f(u, v) = d$ from (3).

The BC proof system from Chapter 5 saturates the set of clauses by forward chaining starting from positive information provided by the congruence closure of assumptions. By contrast, the more common resolution-based methods start with assuming that the goal to be proved is false, and work by backward chaining.

From the formal system, we derive a decision algorithm for logical consequence of equalities from a set of domain-bounded clauses, called the *BC closure algorithm*. The algorithm is based on the congruence closure algorithm, which provides the data structure storing equivalence classes of terms in the domain D, and initiates matching when two classes are merged. In Sect. 5.5, we describe and evaluate our initial implementation of the algorithm.

We intend to use the BC closure algorithm in a new implementation of the CL proof assistant. It should solve or at least mitigate the problems with the current implementation outlined in Par. 1.2.3. The congruence closure part of the BC closure algorithm builds equivalence classes of terms. The internal data structure allows us to display equivalence classes to the user, but does not enforce any particular canonical form, thus solving the problem (i). Moreover, the data structure can record the reasons why a term was added to an equivalence class. This information will allow us to construct explanation for the user and independently checkable proofs ([NO03]). This should solve the problem (ii).

The part of the BC closure algorithm deriving consequences of domain-bounded clauses will solve (inevitably only partially) the problem (iii) from Par. 1.2.3. A large

collection of formal proofs was accumulated during the use of the current CL implementation. Often, lemmas and induction hypotheses are universally quantified clauses. In order to use such a clause in a formal proof, it must be instantiated, i.e., variable-free terms must be substituted for quantified variables. These terms are often already present in the assumptions or goals at the particular point in the proof. Therefore, it may suffice to set the domain to the set of available terms, and run the algorithm. The algorithm is useful even if a new term is needed to instantiate a clause. The user can add the term to the domain, and let the algorithm find suitable terms for other variables. The domain can be also automatically expanded to include terms obtained by rewriting similar to canonization in the current version of CL.

We provide evidence for usefulness of the BC closure algorithm based on a case study in Sect. 5.5. In the case study, the algorithm automatically finished about 50 % of proofs build in the current CL proof assistant. There is, however, room for improvement: The core of the algorithm computing congruence closure can be combined with decision procedures for so-called small theories (e.g., theories of lists, linear arithmetic, partial orderings) using the Nelson-Oppen method ([NO80]). The decision procedures should provide the new proof assistant with built-in knowledge comparable to that of the current proof assistant. The BC closure algorithm automatically finished 37.5 % of proofs which needed manual instantiations in the current version of CL. We consider this a promising result, which should be further improved by combination with decision procedures for small theories.

2 Second-Order Logic for Modular Programming

In this chapter, we present a variation of second-order logic, a new deductive formal system. We then present a subsystem of second-order arithmetic, which, we believe, can serve as a suitable basis for a feature-rich modular programming language.

In Sect. 2.1, we will give a motivation for the use of second-order logic as a means of capturing modularity in programming. We will also outline reasons for introducing our own formal system, describe the intuition behind the system, and give a preliminary example of its use.

Sections 2.2 through 2.6 will deal with pure second-order logic with Henkin semantics, and with our formal system, which is a second-order version of the formal system we presented in [KV09]. Sections from 2.7 will deal with second-order arithmetic. We will define a weak second-order arithmetic theory CL_2, which we intend to use as a foundation for a modular programming language. We will illustrate on a more complex example how CL_2 supports modularization of programs. We will also describe the relationship of CL_2 with standard theories of second-order arithmetic, especially RCA_0.

2.1 Motivation

The current version of CL implements a theory of programming which is based on perhaps the weakest possible theoretical foundation — Peano arithmetic (PA). This theory of programming in PA is developed in the book [Vod01a], which shows how many programming concepts, even a weak kind of functionals, can be expressed in PA. The thesis [Kom09] then builds a large library of verified programs in PA.

Syntactically, CL programs are conjunctions of clauses in the first-order language of PA. Semantically, data structures are natural numbers, and programs are primitive recursive functions or predicates on natural numbers. The semantics of most other programming languages, declarative or imperative, is based on much more complex domains usually containing higher-order functionals.

We will now, in Par. 2.1.1, schematically describe programming and proving in CL. We will point out in Par. 2.1.2 an important part of the design of a programming language, namely modularity, where the first-order logical basis of CL appears to fall short. We will then propose in Par. 2.1.5 a second-order model of modularity. In Par. 2.1.7, we will introduce a formal system with a single rule of inference which integrates proof and theory development. The formal system and the second-order model of modularity will both be developed in detail later in this chapter.

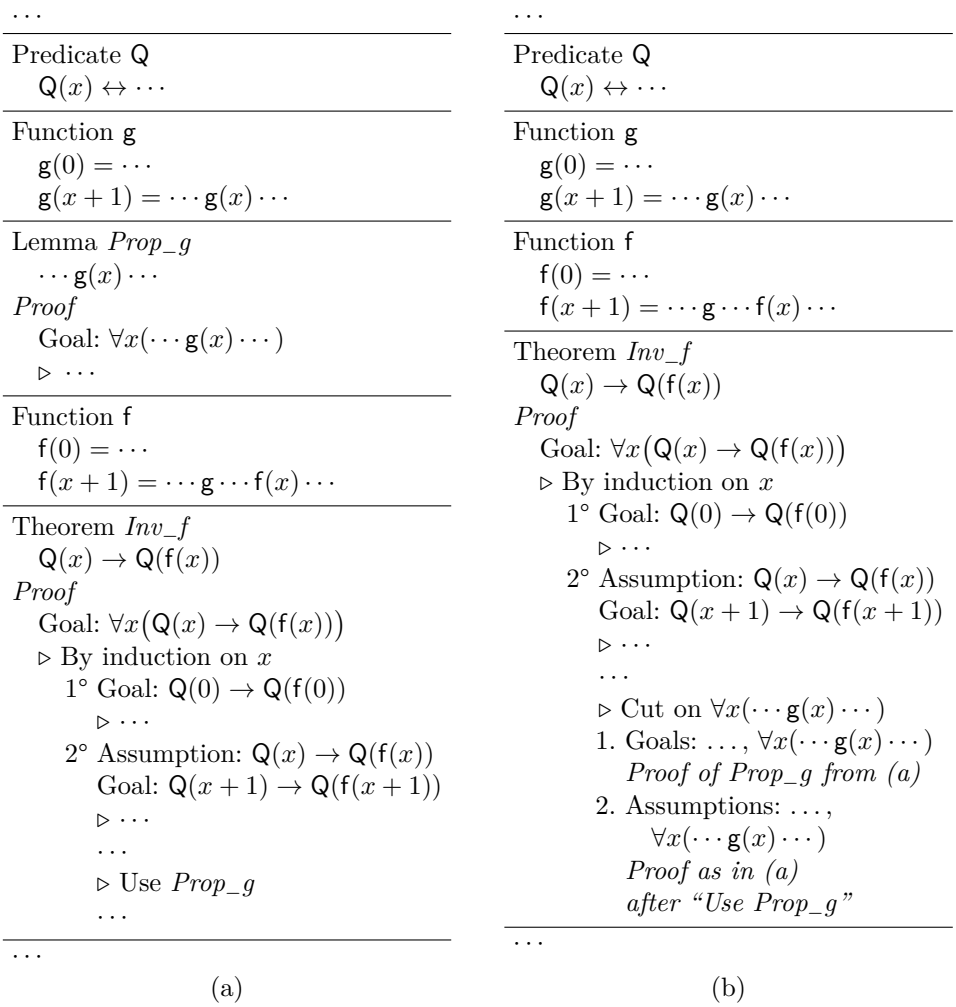


Figure 2.1. Outline of implementation and verification of a program in CL.

2.1.1 Incremental development of verified programs. Let us describe a typical situation during the development of a verified program in the current version of CL. The situation is outlined in Fig. 2.1(a).

A verified program development in CL is organized as a series of components. The most important kinds of components are definitions of functions or predicates, and theorems (and lemmas) with proofs. There are also auxiliary components containing remarks or describing infix notation of some symbols. Components can be interactively added and modified in the CL environment.

From the logical point of view, each function or predicate S is defined in an extension T of Peano arithmetic (PA). The component containing the definition of S further extends T into a theory T_{S} which contains the defining axiom of the symbol S. All subsequent components are then related to T_{S}: Theorems are proved in T_{S}, and definitions are made in T_{S}. All extensions are by definitions. Therefore

by [Sho67, §4.6], all theorems of T_{S} can be equivalently translated to theorems of T, and ultimately to theorems of PA. Thus, logically, definitions are just abbreviations. However, function and predicate definitions are central in the programming point of view. They are the programs which we are developing, and they carry computational meaning.

We have already mentioned in the introduction that in CL, proofs are formalized in a variation of Smullyan's tableau calculus for first-order logic ([Smu68]). In a traditional Hilbert-style formal system, a proof is a sequence of formulas. We can add axioms and formulas derived by inference rules to the end of the sequence until the desired conclusion is derived. In the tableau calculus, a proof is a finite tree in which each node is labeled with a signed formula. Such a tree is called a tableau. The signs attached to formulas denote whether they are *assumptions* which are considered true, or *goals* which are to be proved (or, equivalently, are considered false). A tableau starts with the formula which we want to prove, signed as a goal. Tableau rules allow derivation of new, simpler facts from both assumptions and goals, sometimes splitting the tableau into several branches. The original goal is proved if in each branch, there is an identical formula signed as both an assumption and a goal.

Figure 2.1(a) depicts the following situation: The function f is the main program which we want to verify. It is defined by primitive recursion using an auxiliary function g, which is also recursive. The function f maintains an invariant which is defined as the predicate Q. The main goal of the verification is to prove the theorem *Inv_f*.

The proof is developed by the user interactively, with the help of the CL proof assistant. The assistant informs the user of the current goals and assumptions and simplifies them using built-in knowledge and definitions. In this case, the user instructs the assistant (indicated by ▷) to prove the invariant theorem *Inv_f* by induction. The proof is split into two independent branches, one for the base case, and another for the inductive step. Since f is defined using g, we need an auxiliary property of g. This property is asserted and proved as the lemma *Prop_g*, which is inserted just below the definition of g. We then use the lemma *Prop_g* in the course of the proof of the inductive step of *Inv_f*.

2.1.2 Modularity concerns. The development illustrated in Fig. 2.1(a) exposes auxiliary definitions and lemmas to any components lying below *Inv_f*. This is not unusual in mathematics, where definitions, lemmas, and main theorems are usually presented linearly. On the other hand, in programming, it is usual to organize programs into *modules*. A module hides (*encapsulates*) implementation details of a part of a program, and exposes only a higher-level *interface* to other parts of the program. Even in mathematics, simple auxiliary definitions are sometimes made within proofs of theorems if they are not needed anywhere else. Moreover, many languages allow modules to be *parameterized*. Parameterized modules are a powerful tool for abstraction of structurally similar parts of a program.

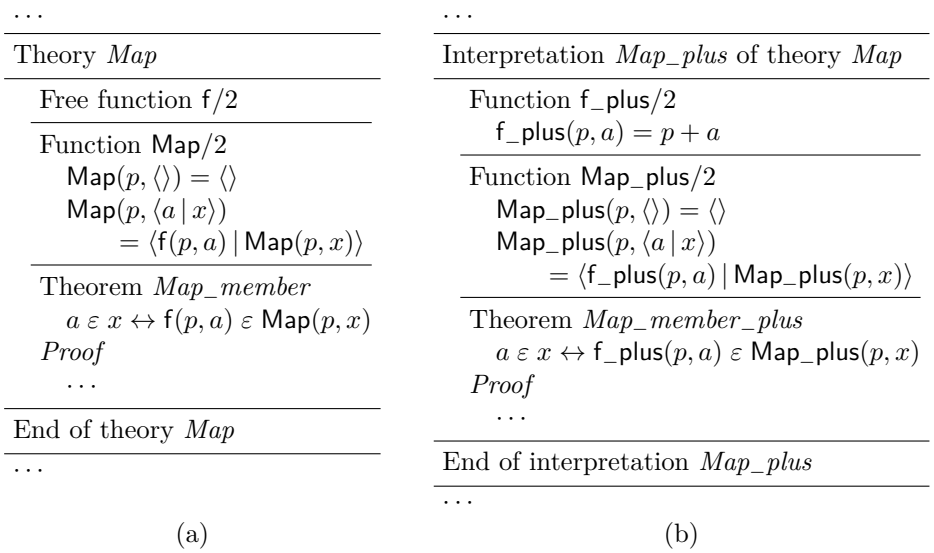


Figure 2.2. Example of using a theory in CL as a parameterized module.

2.1.3 Encapsulation in CL. The current version of CL offers only a very limited form of encapsulation. It is possible to use a tableau rule called a *cut on a formula A* anywhere in a proof in order to split the proof into two branches: one in which A is a goal, and one in which A is an assumption. The cut rule allows embedding of a lemma into the proof of a theorem. In the example from Fig. 2.1(a), instead of using lemma *Prop_g* in the proof of theorem *Inv_f*, we can cut on its formula $\forall x(\cdots \mathsf{g}(x) \cdots)$, and insert lemma's proof directly into the proof of the theorem. The result of this transformation can be seen in Fig. 2.1(b).

However, there is no analogous mechanism for embedding definitions into other definitions or into proofs.

2.1.4 Parameterized modules in CL. There is an powerful mechanism for construction of parameterized modules in the current version of CL. It is possible to define theories and *interpret* them ([Sho67, §4.7]) in PA. In CL, a theory is a component containing other components. Two special kinds of components are allowed in theories: free symbols and axioms. A free symbol extends the language of the theory with a new function or predicate symbol, but does not extend the theory with any defining axiom. An axiom is simply a formula which is considered true within the theory. It may contain the free symbols. Function and predicate definitions and theorems in the theory may also contain the free symbols. Proofs of theorems may use axioms.

When a theory is viewed as a parameterized module, free symbols are parameters of the module. Axioms specify constraints that must be satisfied by parameters. Defined functions and predicates, as well as proved theorems are parameterized by free symbols.

An example of a theory is depicted in Fig. 2.2(a). The theory *Map* contains a

free binary function symbol f and no axioms. The function Map applies f with a parameter p to all members of a list (a finite sequence, see Par. 2.3.2 below, encoded as a natural number). Map is defined by list recursion (which is reducible to primitive recursion). A theorem *Map_member* linking membership (ε) in the input list x and in the output list $\mathsf{Map}(p, x)$ is then proved.

An instance of a parameterized module can be created as an interpretation of a theory. In the interpretation, each free symbol of the theory is given a definition, and each axiom of the theory is proved as a theorem. We then automatically obtain instances of the functions, predicates, and theorems in the theory. A possible interpretation *Map_plus* of the theory *Map* is depicted in Fig. 2.2(b).

Theories offer a kind of parameterization seldom seen in other programming languages. Axioms in a theory can place arbitrary PA-expressible requirements on parameters. These requirements can go far beyond typing constraints allowed in functional programming languages such as Haskell ([PJ03]) with a Hindley-Milner type system. Typing in most other languages is even weaker than in Haskell. Also, theorems in a theory can express properties of parameterized functions defined in the theory in much greater detail than typing.

However, theories and interpretations offer no way of hiding implementation details. An interpretation exposes interpreted versions of all components of the interpreted theory, except for lemmas. Another disadvantage is a rigid naming convention for components of an interpretation: a suffix is attached to names of all components of the interpreted theory.

2.1.5 Modules in second-order logic. We have seen in the previous two paragraphs that CL offers a powerful way to create parameterized modules, but is unable to hide their internal details, especially definitions. It is impossible to hide a definition of a function and expose only its proved properties.

The solution to the hiding problem proposed in this chapter employs second-order logic (Sect. 2.2) and a weak theory of second-order arithmetic (Sect. 2.7) instead of PA. In second-order logic, quantification over functions and predicates (i.e., second-order quantification) is possible. We can replace CL theories with theorems with second-order quantification. For example, the theory *Map* from Fig. 2.2(a) can be replaced with the second-order theorem

$$\begin{aligned}\forall f \exists Map \forall x \forall a \forall p \big(& Map(p, \langle\rangle) = \langle\rangle \\ & \wedge\, Map(p, \langle a \,|\, x\rangle) = \langle f(p, a) \,|\, Map(p, x)\rangle \\ & \wedge\, (a \;\varepsilon\; x \leftrightarrow f(p, a) \;\varepsilon\; Map(p, x))\big)\end{aligned} \tag{Map_ex}$$

asserting existence of a function *Map* with given properties for each function f. Note that even the recursive definition is now just a property of the function *Map*. If desired, we can remove the recursive definition from the theorem, leaving only the property called *Map_member* in Fig. 2.2(a):

$$\forall f \exists Map \forall x \forall a \forall p (a \;\varepsilon\; x \leftrightarrow f(p, a) \;\varepsilon\; Map(p, x)). \tag{Map_ex'}$$

The theorem *Map_ex′* gives only a high-level specification of the function *Map*. The recursive definition of *Map* will be hidden in the proof of *Map_ex′*.

Note that in second-order logic with a suitable theory of second-order arithmetic, existence of each primitive recursive function is provable as a theorem. For instance,

$$\exists Fact \forall n(\, Fact(0) = 0 \\ \wedge\, Fact(n+1) = (n+1) \cdot Fact(n)).$$

Proofs in second-order logic are not much different from those in first-order logic. Rules dealing with second-order quantifiers are essentially the same as for their first-order counterparts. The power of second-order logic lies in the axioms of existence of second-order objects (functions and predicates) contained in a particular second-order theory.

In our case, we want to keep to the spirit of the current version of CL. CL allows definitions of recursive functions and predicates only. (It was originally planned to allow primitive recursive functions only, but the necessary restrictions were not enforced in the actual implementation.) Therefore, the particular second-order theory CL_2 that we propose for the new implementation of CL allows existence of only recursive functions and predicates. The theory will be detailed in Sect. 2.7. Further examples of using CL_2 for modular development will be given in Sect. 2.8.

2.1.6 Theory extensions and proofs. If we prove a theorem of the form $\exists f A$, then we can extend the theory CL_2 with a new function symbol f and an axiom $A[f/\mathsf{f}]$ (without significantly altering the power of the theory, see Cor. 2.5.5). For example, suppose that we have proved

$$\exists Map \forall x \forall a \forall p(a \;\varepsilon\; x \leftrightarrow (p+a) \;\varepsilon\; Map(p,x)) \qquad (Map\_ex'_+)$$

by a simple instantiation of 2.1.5(Map_ex') with standard arithmetic $+$ for f. We can then extend CL_2 with a new function symbol Map_+ and a new axiom

$$\forall x \forall a \forall p(a \;\varepsilon\; x \leftrightarrow p + a \;\varepsilon\; \mathsf{Map}_+(p,x)).$$

This operation is similar to incremental extensions of PA in the current version of CL by definitions of functions and predicates, described in Par. 2.1.1.

However, we have realized that we do not need to introduce a special operation for theory extension. As we have noted already in Par. 2.1.2, a lemma can be embedded into a proof of a theorem using the tableau cut rule. In second-order logic, embedding of definitions becomes possible, too. We will then obtain the following logical model of verified programming:

(a) An incremental development of a verified program can be viewed as a proof with no initial goal.

(b) In order to prove a new theorem, we apply the cut rule to its formula A. We then prove A as a goal in one branch, and assume that A holds in the other branch.

(c) In order to introduce a new function whose existence has been proved as a theorem $\exists f A$, we use the eigen-constant tableau rule (Rule D in [Smu68]): if $\exists f A$ is an assumption, then we can derive a new assumption $A[f/\mathsf{f}]$ for a new function symbol f with the same arity (number of arguments) as the function variable f.

We will describe the unification of theory extensions and proofs in more detail in Sect. 2.5.

2.1.7 Valuation trees as proofs. The unification of theory extensions and proofs described in the previous paragraph has led us to observation that we can build a complete formal system for second-order logic on the cut rule only.

Smullyan's tableau calculus (as well as related Gentzen's sequent calculus and natural deduction) has many rules, and no axioms. By contrast, the traditional Hilbert-style calculus has one propositional rule (modus ponens), one quantification rule, and infinitely many axioms. Our cut-based formal system has only one rule, and infinitely many axioms. However, unlike the Hilbert-style calculus, our system allows intuitive proofs in a style similar to tableaux or sequent calculus.

We describe the formal system in Sect. 2.3, and relate it to tableaux and sequent calculus in Sect. 2.4. We expect that using our formal system in the proof assistant of the new implementation of CL will bring practical advantages. An internal data structure supporting the single-rule formal system should be simpler than the current structure supporting the many-rule tableau calculus. This simplification should improve modularity and extensibility of the proof assistant.

A cut is also an elegant way of integrating the proof-building and the automatic part of a proof assistant. If the automatic part derives a new fact A from current assumptions, it can add it using a cut on A, and provide a proof in the branch where A is a goal. The user then continues proving in the branch where A is an assumption. If interested, the user can review the automatic proof of A.

We will now illustrate the use of our formal system with a simple propositional example. Let us informally introduce the notion of a sequent. When we write $A_1, A_2, \ldots, A_m \Rightarrow G_1, G_2, \ldots, G_n$, we mean that we assume that *all* of the formulas A_1, A_2, …, A_m hold, and we want to prove any *one* of the formulas G_1, G_2, …, G_n. The expression $A_1, \ldots, A_m \Rightarrow G_1, \ldots, G_n$ is called a *sequent*. It is often more intuitive to see a sequent as a description of a situation in a proof by contradiction, i.e., we assume that *all* formulas A_1, …, A_m hold, and that *no* formula G_1, …, G_n holds.

We would like to prove Pierce's law

$$P \equiv \big(((Q \to R) \to Q) \to Q\big),$$

i.e., we have $\Rightarrow P$ in the sequent notation. Let us analyze the antecedent $(Q \to R) \to Q$ of the implication P. Assume first that $\Rightarrow ((Q \to R) \to Q), P$. This sequent is an instance of a schema $\Rightarrow A, (A \to B)$, which is obviously contradictory: It cannot be the case that both the antecedent A and the implication $A \to B$ do not hold, since if A does not hold, then $A \to B$ holds. So, in this case, the proof is trivial. In the other case, we have $((Q \to R) \to Q) \Rightarrow P$, where we do not see any obvious contradiction. Further analysis is necessary.

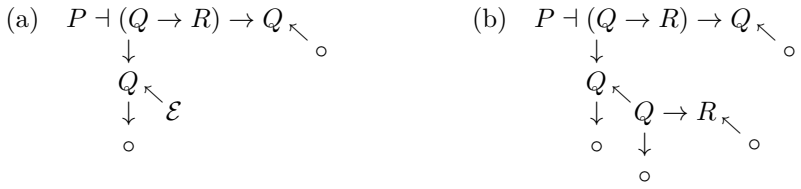


Figure 2.3. A valuation tree proving Pierce's Law.

Let us now analyze the consequent Q of the implication P. In the first case, $Q, ((Q \to R) \to Q) \Rightarrow P$, we have a contradiction. This time, the contradictory sequent contains an instance of a schema $B \Rightarrow (A \to B)$. It cannot be the case that an implication $A \to B$ does not hold, and its consequent B holds. So the other option remains: $((Q \to R) \to Q) \Rightarrow Q, P$. Since we see no obvious contradiction here, further case analysis is necessary again.

Before continuing, we will introduce a tree notation for our proofs by cuts (or, by case analysis of truth values of formulas). The proof described so far will be written as shown in Fig. 2.3(a). The intended reading of $P \dashv \mathcal{D}$ is "P is proved by $\mathcal{D}$". Each case analysis from the above informal discussion is represented by an internal node of the tree in the figure. Internal nodes are labeled with the analyzed formulas. A left-hand child of an internal node describes the proof under the assumption that the label holds. A right-hand child describes the proof where we assume that the label does not hold. Each path leading from the root of the tree to a node thus determines a sequent. Each leaf $\circ$ expresses that the sequent determined by the path to the leaf is contradictory. In order to finish the proof, we need to find a tree $\mathcal{E}$. The path to the root of $\mathcal{E}$ determines the sequent $((Q \to R) \to Q) \Rightarrow Q, P$.

In order to see that this sequent is contradictory, we will analyze the formula $Q \to R$. In case $Q \to R$ holds, we have $(Q \to R), ((Q \to R) \to Q) \Rightarrow Q, P$, which is obviously contradictory, since Q does not hold, but both $Q \to R$ and $(Q \to R) \to Q$ do hold. This contradiction is an instance of a schema $A, (A \to B) \Rightarrow B$. In case $Q \to R$ does not hold, we have a contradiction $((Q \to R) \to Q) \Rightarrow (Q \to R), Q, P$, since Q does not hold, and $Q \to R$ does not hold either. This is an instance of the schema $\Rightarrow A, (A \to B)$ which we have already seen at the beginning of the proof.

Since both cases are contradictory, we conclude that assuming that P does not hold leads to a contradiction, and thus P is proved. The tree of the complete proof is shown in Fig. 2.3(b).

The proof tree describes analysis of truth *values* of formulas in its nodes. We therefore call it a *valuation tree*. In the above example, we have used three schemas of obviously contradictory sequents: $\Rightarrow (A \to B), A$; $B \Rightarrow (A \to B)$; and $A, (A \to B) \Rightarrow B$. These schemas are the axioms of implication in our formal system. There are similar axioms for other propositional connectives and also for quantifiers.

2.2 Second-Order Logic with Functions and Predicates

Standard second-order logic (see, e.g., [Sim99]) is concerned with objects of two sorts: first-order objects (e.g., numbers), and second-order objects, which are sets of first-order objects. The version of second-order logic presented in this section deals with many sorts of second-order objects, namely, functions and predicates of all possible arities (numbers of arguments). Although this logic is slightly more complicated, our choice is driven by the intended application. Programs are naturally viewed as functions, and their properties as predicates.

Syntax of Second-Order Logic

2.2.1 Second-order languages. A *second-order language* $\mathcal{L}$ is a many-sorted language consisting of variables and constants. There is a countable infinite set of *first-order variables* denoted by x, y, z, and a countable (finite or infinite) set of *first-order constants* denoted by c, d. For each $n > 0$, there is a countable infinite set of *function variables of arity n* denoted by f, g, a countable infinite set of *predicate variables of arity n* denoted by P, R, a countable set of *function constants of arity n* denoted by f, g, and a countable set of *predicate constants of arity n* denoted by P, R.

If the sort of a variable or a constant is irrelevant, it will be denoted by X or by C respectively. A constant not present in a language is called *new*. We denote by $\mathcal{L}[\mathcal{C}]$ the *extension* of the language $\mathcal{L}$ with constants in the countable set $\mathcal{C}$ which are all new. We assume, that for each language and each sort there is an infinite recursive set of new constants.

2.2.2 Terms, formulas, theories. *Expressions* of a language $\mathcal{L}$ are finite sequences built from variables and constants of $\mathcal{L}$, the *equality symbol* "=", *propositional connectives* "$\neg$", "$\wedge$", "$\vee$", "$\rightarrow$", *quantifiers* "$\forall$", "$\exists$", parentheses, and commas. For two expressions e_1 and e_2, we write $e_1 \equiv e_2$ if they are identical as sequences of symbols.

First-order terms of $\mathcal{L}$ (denoted by s, t) are the least set of expressions containing all first-order variables and constants, and closed under the rule: if f is an n-ary function variable or constant of $\mathcal{L}$, and t_1, t_2, ..., t_n are numeric terms, then $f(t_1, t_2, \ldots, t_n)$ is a first-order term.

Atomic formulas of $\mathcal{L}$ are *equalities* $t_1 = t_2$ for all first-order terms t_1 and t_2 of $\mathcal{L}$, and *predicate applications* $P(t_1, \ldots, t_n)$ for all $n \geq 0$, predicate variables and constants P of arity n, and first-order terms t_1, ..., t_n of $\mathcal{L}$.

Formulas of $\mathcal{L}$ (denoted by A, B, C) are the least set of expressions containing all atomic formulas and closed under the rules:

(i) if A and B are formulas, then also *negation* $\neg A$, *conjunction* $(A \wedge B)$, *disjunction* $(A \vee B)$, and *implication* $(A \rightarrow B)$ are formulas;

(ii) if A is a formula and X is a variable, then *universal quantification* $\forall X A$ and *existential quantification* $\exists X A$ are formulas.

We will use the standard notions of free and bound occurrences of variables, and *substitution* $e[X/t]$ of all free occurrences of a variable X in a term or formula e.

Natural sorting constraints apply to substitutions: For first-order variables, t can be any first-order term, for function or predicate variables, t can only be a variable or constant of the same sort as X.

If A is a formula, and $X_1, \ldots, X_n$ are all variables that occur freely in A, then the sentence $\forall X_1 \cdots \forall X_n A$ is called the *universal closure* of A.

A formula or a term with no free variables is called *closed*. Closed formulas of $\mathcal{L}$ are called *sentences* of $\mathcal{L}$. A set of sentences of $\mathcal{L}$ is called a *theory* in $\mathcal{L}$. A member of a theory is called an *axiom*. We will use letters T, S to denote theories, and Greek letters Γ, Δ, Π to denote finite theories. If a language $\mathcal{L}'$ is an extension of a language $\mathcal{L}$, a T' a theory in $\mathcal{L}'$ is called an *extension* of a theory T in $\mathcal{L}$ if $T \subseteq T'$.

2.2.3 Abbreviations. For theories T_1 and T_2, we will write T_1, T_2 instead of $T_1 \cup T_2$, and T_1, A or A, T_1 instead of $T_1 \cup \{A\}$. We will, however, avoid this practice in cases where it might lead to confusion.

If $n \geq 0$ is clear from the context, we abbreviate a sequence of terms $t_1, \ldots, t_n$ to $\vec{t}$. We can then write $\vec{t} \in S$ instead of "$t_i \in S$ for all $i = 1, \ldots, n$", $f(\vec{t})$ instead of "$f(t_1, \ldots, t_n)$", $\forall \vec{x}$ instead of "$\forall x_1 \cdots \forall x_n$" (and similarly for $\exists$), and $\langle \vec{t} \rangle$ instead of "$\langle t_1, \ldots, t_n \rangle$". We abbreviate $t_1 = s_1, \ldots, t_n = s_n$ to $\vec{t} = \vec{s}$ on both the meta-level ("=" representing equality) and the object level ("=" representing the equality symbol).

In formulas, we usually omit the outermost parentheses. Conjunction has higher priority than disjunction, which has higher priority than implication, so we abbreviate $((A \wedge B) \vee C) \to D$ to $A \wedge B \vee C \to D$. Propositional connectives of equal priority associate to the right, so $A \to (B \to C)$ abbreviates to $A \to B \to C$. The left-hand side of an implication is called the *antecedent*, and the right-hand side is called the *consequent*. *Equivalence* $(A \leftrightarrow B)$ is an abbreviation of $(A \to B) \wedge (B \to A)$, and has the same priority as implication.

Henkin Semantics of Second-Order Logic

2.2.4 Structures and expansions. A *structure* $\mathcal{M}$ for a second-order language $\mathcal{L}$ is a quadruple $(M, \{\mathcal{F}_i\}_{i=1}^{\infty}, \{\mathcal{R}_i\}_{i=1}^{\infty}, (\cdot)^{\mathcal{M}})$ where M is a set called the (first-order) *domain* of $\mathcal{M}$, and for each $n > 0$, $\mathcal{F}_n$ is some set of n-ary functions on M (i.e., $\mathcal{F}_n \subseteq M^{M^n}$), and $\mathcal{R}_n$ some set of n-ary relations on M (i.e., $\mathcal{R}_n \subseteq \mathcal{P}(M^n)$). The function $(\cdot)^{\mathcal{M}}$ assigns to each first-order constant c of $\mathcal{L}$ an element $\mathsf{c}^{\mathcal{M}}$ of M, to each n-ary function constant f a function $\mathsf{f}^{\mathcal{M}} \in \mathcal{F}_n$, and to each n-ary predicate constant R a relation $\mathsf{R}^{\mathcal{M}} \in \mathcal{R}_n$. The sets M, $\mathcal{F}_n$, $\mathcal{R}_n$ also serve as ranges of variables of the respective sorts. We require that M, $\mathcal{F}_n$, $\mathcal{R}_n$ are non-empty, and $M \cap \mathcal{R}_n = \emptyset$ for all $n > 0$.

Given a structure $\mathcal{M}$ for $\mathcal{L}$ and a countable set of constants $\mathcal{C}$, a structure $\mathcal{M}'$ is an *expansion of* $\mathcal{M}$ *to* $\mathcal{L}[\mathcal{C}]$ if $\mathcal{M}'$ is a structure for $\mathcal{L}[\mathcal{C}]$ such that its domain, and its sets functions and relations are the same as those of $\mathcal{M}$, and the assignment $(\cdot)^{\mathcal{M}'}$ coincides with $(\cdot)^{\mathcal{M}}$ on all constants of $\mathcal{L}$. The structure $\mathcal{M}$ is then called the *reduct*

of $\mathcal{M}'$ to $\mathcal{L}$. For a new constant C and an object m of the corresponding sort from $\mathcal{M}$, we denote by $\mathcal{M}[\mathsf{C} \mapsto m]$ the expansion of $\mathcal{M}$ to $\mathcal{L}[\mathsf{C}]$ such that $\mathsf{C}^{\mathcal{M}[\mathsf{C}\mapsto m]} = m$.

2.2.5 The satisfaction relation and logical consequence. The assignment $(\cdot)^{\mathcal{M}}$ can be extended to all closed terms of $\mathcal{L}$ by structural induction on terms so that $(\mathsf{f}(t_1, \ldots, t_n))^{\mathcal{M}} = \mathsf{f}^{\mathcal{M}}(t_1^{\mathcal{M}}, \ldots, t_n^{\mathcal{M}})$.

Satisfaction of a sentence A in a structure $\mathcal{M}$, in writing $\mathcal{M} \vDash A$, is the smallest relation with the following properties:

(i) $\mathcal{M} \vDash s = t$ iff $s^{\mathcal{M}} = t^{\mathcal{M}}$;
(ii) $\mathcal{M} \vDash \mathsf{R}(t_1, \ldots, t_n)$ iff $(t_1^{\mathcal{M}}, \ldots, t_n^{\mathcal{M}}) \in \mathsf{R}^{\mathcal{M}}$;
(iii) $\mathcal{M} \vDash \neg A$ iff $\mathcal{M} \nvDash A$;
(iv) $\mathcal{M} \vDash A \wedge B$ iff $\mathcal{M} \vDash A$ and $\mathcal{M} \vDash B$;
(v) $\mathcal{M} \vDash A \vee B$ iff $\mathcal{M} \vDash A$ or $\mathcal{M} \vDash B$;
(vi) $\mathcal{M} \vDash A \to B$ iff $\mathcal{M} \vDash B$ whenever $\mathcal{M} \vDash A$;
(vii) $\mathcal{M} \vDash \forall X A$ iff $\mathcal{M}[\mathsf{C} \mapsto m] \vDash A[X/\mathsf{C}]$ for any constant C of the same sort as X not present in the language of $\mathcal{M}$ and for all m of the respective sort from $\mathcal{M}$;
(viii) $\mathcal{M} \vDash \exists X A$ iff $\mathcal{M}[\mathsf{C} \mapsto m] \vDash A[X/\mathsf{C}]$ for any constant C of the same sort as X not present in the language of $\mathcal{M}$ and some m of the respective sort from $\mathcal{M}$.

We say a structure $\mathcal{M}$ is a *model* of a theory T, in writing $\mathcal{M} \vDash T$, if $\mathcal{M}$ satisfies every sentence $A \in T$. A sentence A is a *logical consequence* of a theory T, in writing $T \vDash A$, if every model of T satisfies A. We also say that T *logically implies* A. For two theories S and T, we write $T \vDash S$ if $T \vDash B$ for all $B \in S$. We say a sentence A is *valid* if $\emptyset \vDash A$.

2.3 Valuation Trees as Proofs

2.3.1 Introduction to valuation trees. A *valuation tree* (denoted by $\mathcal{D}$, $\mathcal{E}$) is a tree with each node either a leaf or an internal node with two ordered successors. Leaves are designated by the symbol $\circ$. Internal nodes are labeled with sentences, and written down as

$$\begin{array}{l} A \\ \downarrow \quad \mathcal{E} \\ \mathcal{D} \end{array}$$

where A is the label, $\mathcal{D}$ is the left successor, and $\mathcal{E}$ the right successors.

Intuitively, an internal node assigns the sentence A the value "true" in $\mathcal{D}$, and the value "false" in $\mathcal{E}$. We prefer a positive interpretation: the sentence A is an assumption in $\mathcal{D}$, and a goal to be proved in $\mathcal{E}$. In order to discern these two roles of a sentence, we define signed sentences in Par. 2.3.3. There are other calculi in which sentences also have two roles, e.g., Smullyan's calculus of signed tableaux, or Gentzen's sequent calculus where all assumptions are collected in the antecedent Γ and goals in the consequent Δ of a sequent $\Gamma \Rightarrow \Delta$.

Each simple path from the root of a valuation tree to a leaf is uniquely associated with a finite sequence of signed formulas. In fact, the whole tree is uniquely defined

by a finite set of such sequences. We describe these sequences and sets in Par. 2.3.4 using a Prolog-like notation introduced in Par. 2.3.2.

2.3.2 Sequences. A *sequence* p is a function from an initial subset $\{ i \mid i < n \}$ of natural numbers to some set. The *length* of a sequence p is its cardinality $|p|$. The *empty sequence*, denoted by $\langle\rangle$, is the empty set. We define $\langle a \,|\, p\rangle$ as the sequence of length $|p| + 1$ such that $\langle a \,|\, p\rangle(0) = a$ and $\langle a \,|\, p\rangle(i + 1) = p(i)$ for all $i < |p|$. We abbreviate $\langle a_0 \,|\, \langle a_1 \,|\, \cdots \langle a_n \,|\, \langle\rangle\rangle \cdots\rangle\rangle$ to $\langle a_0, a_1, \ldots, a_n\rangle$.

For sequences p and q, their *concatenation* $p +\!\!+ q$ is defined as the sequence of length $|p| + |q|$ such that $(p +\!\!+ q)(i) = p(i)$ for all $i < |p|$ and $(p +\!\!+ q)(i + |p|) = q(i)$ for all $i < |q|$. Note that we have $\langle\rangle +\!\!+ q = q +\!\!+ \langle\rangle = q$ and $\langle a \,|\, p\rangle +\!\!+ q = \langle a \,|\, p +\!\!+ q\rangle$.

A sequence p is a *prefix* of a every sequence $p +\!\!+ q$, and a *proper prefix* of every $p +\!\!+ \langle a \,|\, q\rangle$. Note that since sequences are sets of ordered pairs $(i, p(i))$, p is a (proper) prefix of q iff $p \subseteq q$ ($p \subset q$). We call each $p +\!\!+ q$ a *successor* of p, each $p +\!\!+ \langle a \,|\, q\rangle$ a *proper successor* of p, and each $p +\!\!+ \langle a\rangle$ an *immediate successor* of p.

2.3.3 Signed sentences, valuation sequences, and sequents. A *signed sentence* of a language $\mathcal{L}$ is a sentence of $\mathcal{L}$, or a sentence of $\mathcal{L}$ suffixed with the symbol $*$.

We call a sequence of signed sentences of a language $\mathcal{L}$ a *valuation sequence* in $\mathcal{L}$, and a finite set of signed sentences a *sequent* in $\mathcal{L}$. Within valuation sequences and sequents, sentences without $*$ are called *assumptions*, and signed sentences with $*$ are called *goals*.

For finite sets Γ and Δ of (unsigned) sentences, we define

$$(\Gamma \Rightarrow \Delta) = \{ A \mid A \in \Gamma \} \cup \{ A* \mid A \in \Delta \}.$$

Each sequent can be uniquely written as $\Gamma \Rightarrow \Delta$ for suitable Γ and Δ, called respectively the *antecedent* and the *consequent* of the sequent.

The image $\{ p(i) \mid i < |p| \}$ of a valuation sequence p is a sequent, called *the sequent of* p and denoted by $\Gamma_p \Rightarrow \Delta_p$. For a sequence p, we write Γ_p to denote the antecedent of $\Gamma_p \Rightarrow \Delta_p$, and Δ_p to denote its consequent, even without mentioning the sequent itself.

We say that a structure $\mathcal{M}$ for $\mathcal{L}$ *satisfies a sequent* $\Gamma \Rightarrow \Delta$ in $\mathcal{L}$, and write $\mathcal{M} \models \Gamma \Rightarrow \Delta$, if $\mathcal{M}$ satisfies some $A \in \Delta$ whenever $\mathcal{M} \models \Gamma$. That is, a structure satisfies a sequent if it either does not satisfy some assumption, or it satisfies all assumptions and at least one goal. A theory T *logically implies* a sequent $\Gamma \Rightarrow \Delta$, in writing $T \models \Gamma \Rightarrow \Delta$, if each structure satisfying T satisfies $\Gamma \Rightarrow \Delta$. A sequent is *valid* if $\emptyset \models \Gamma \Rightarrow \Delta$.

Note that if a structure $\mathcal{M}$ for $\mathcal{L}$ satisfies $\Gamma \Rightarrow \Delta$, then it satisfies $\Gamma, \Gamma' \Rightarrow \Delta, \Delta'$ for all Γ', Δ' in $\mathcal{L}$.

2.3.4 Valuation trees. A *valuation tree* $\mathcal{D}$ *for a valuation sequence* p in a language $\mathcal{L}$ is a non-empty finite set of valuation sequences in some expansion $\mathcal{L}[\mathcal{C}]$ of $\mathcal{L}$ satisfying:

(a) $\mathcal{D}$ is closed under prefixes ($q \in \mathcal{D}$ whenever $q \subseteq r \in \mathcal{D}$);
(b) each *node* $q \in \mathcal{D}$ is either
  (i) a *leaf* with no proper successor in $\mathcal{D}$, or
  (ii) an *internal node* with exactly two immediate successors in $\mathcal{D}$: $q +\!\!+ \langle A\rangle$ and $q +\!\!+ \langle A*\rangle$ for some sentence A in $\mathcal{L}$, which is called the *label* of q.
(c) for each node $q +\!\!+ \langle A\rangle \in \mathcal{D}$, the sequence $p +\!\!+ q +\!\!+ \langle A\rangle$ is *regular for* $\mathcal{C}$: if a constant $\mathsf{C} \in \mathcal{C}$ occurs in A but not in q, there is a formula B such that $A \equiv B[X/\mathsf{C}]$ and $\exists X B \in \Gamma_p, \Gamma_q$, or $A \equiv \neg B[X/\mathsf{C}]$ and $\forall X B \in \Delta_p, \Delta_q$.

Note that $\mathcal{D}$ is indeed a tree in the set-theoretical sense, i.e., a set of sequences partially well-ordered by the proper prefix relation $\subset$.

Note that each node $q +\!\!+ \langle A\rangle$ of a valuation tree introduces at most one new constant. If two different constants $\mathsf{C}_1, \mathsf{C}_2 \in \mathcal{C}$ occur in A then C_2 occurs in any B such that $A \equiv B[X/\mathsf{C}_1]$ or $A \equiv \neg B[X/\mathsf{C}_1]$. If C_1 does not occur in q, we have $\exists X B \in \Gamma_p, \Gamma_q$ or $\forall X B \in \Delta_p, \Delta_q$. Thus C_2 occurs already in q.

2.3.5 Notation for valuation trees. The notation of valuation trees from Par. 2.3.1 can be formalized as follows:

$$\circ = \{\langle\rangle\}; \qquad \begin{matrix} A \\ \downarrow \\ \mathcal{D} \end{matrix}\!\!\nwarrow \mathcal{E} = \{\langle\rangle\} \cup \{\, \langle A \,|\, p\rangle \mid p \in \mathcal{D}\,\} \cup \{\, \langle A* \,|\, p\rangle \mid p \in \mathcal{E}\,\}.$$

Each valuation tree $\mathcal{D}$ can be constructed from $\circ$ and $\begin{matrix} \cdot \\ \downarrow \\ \cdot \end{matrix}\!\!\nwarrow \cdot$. Since $\mathcal{D}$ is non-empty, finite, and closed under prefixes, it contains $\langle\rangle$. If it has no other element, then $\mathcal{D} = \circ$. Otherwise, $\langle\rangle$ has a label A and exactly two immediate successors $\langle A\rangle$ and $\langle A*\rangle$ in $\mathcal{D}$, and any other $p \in \mathcal{D}$ is a successor of one of them, so we have

$$\mathcal{D} = \begin{matrix} A \\ \downarrow \\ \{\, p \mid \langle A \,|\, p\rangle \in \mathcal{D}\,\} \\ \{\, p \mid \langle A* \,|\, p\rangle \in \mathcal{D}\,\} \end{matrix}\!\!\nwarrow \{\, p \mid \langle A* \,|\, p\rangle \in \mathcal{D}\,\}.$$

2.3.6 Closed sequents and sequences. Let T be a theory in a language $\mathcal{L}$. We say that the sequents in Fig. 2.4 in any language $\mathcal{L}[\mathcal{C}]$ are *simply T-closed*. Note that P in (Repl) is an atomic sentence, i.e., an equality or an application of a predicate constant. Also note that the sorts of X and t in instantiation sequents must be the same, or the substitution would be undefined. The existential sequents $(\exists_1)$, $(\exists_{f,n})$, $(\exists_{R,n})$ express existence of an object of each sort; the first-order variables $x_1, \ldots, x_n$ must be mutually distinct.

A valuation sequence p in a language $\mathcal{L}[\mathcal{C}]$ is *$T, \mathcal{L}$-closed* if it satisfies one of the following conditions:

(i) $\Gamma_p \Rightarrow \Delta_p$ is a superset of a simply T-closed sequent, or
(ii) $p = q +\!\!+ \langle A[X/\mathsf{C}]*\rangle +\!\!+ r$ and $\exists X A \in \Gamma_q$ for some $\mathsf{C} \in \mathcal{C}$, A, q in which C does not occur, and some r, or
(iii) $p = q +\!\!+ \langle \neg A[X/\mathsf{C}]*\rangle +\!\!+ r$ and $\forall X A \in \Delta_q$ for some $\mathsf{C} \in \mathcal{C}$, A, q in which C does not occur, and some r.

If p is not $T, \mathcal{L}$-closed, it is *$T, \mathcal{L}$-open*.

Propositional sequents

$$\begin{array}{llll}
(\mathrm{Ax}) & A \Rightarrow A & (\mathrm{L}\neg) & \neg A, A \Rightarrow \quad (\mathrm{R}\neg) \quad \Rightarrow \neg A, A \\
(\mathrm{L}\wedge_1) & A \wedge B \Rightarrow A & (\mathrm{L}\wedge_2) & A \wedge B \Rightarrow B \quad (\mathrm{R}\wedge) \quad A, B \Rightarrow A \wedge B \\
(\mathrm{L}\vee) & A \vee B \Rightarrow A, B & (\mathrm{R}\vee_1) & A \Rightarrow A \vee B \quad (\mathrm{R}\vee_2) \quad B \Rightarrow A \vee B \\
(\mathrm{L}\to) & A \to B, A \Rightarrow B & (\mathrm{R}\to_1) & \Rightarrow A \to B, A \quad (\mathrm{R}\to_2) \quad B \Rightarrow A \to B
\end{array}$$

Equational sequents

$$(\mathrm{Refl}) \;\Rightarrow t = t \qquad (\mathrm{Repl}) \;\; s = t, P[x/s] \Rightarrow P[x/t] \quad \text{for atomic } P$$

Instantiation sequents

$$(\mathrm{R}\exists) \;\; A[X/t] \Rightarrow \exists X A \qquad (\mathrm{L}\forall) \;\; \forall X A \Rightarrow A[X/t]$$

Existential sequents

$$\begin{array}{ll}
(\exists_1) & \Rightarrow \exists x (x = x) \\
(\exists_{f,n}) & \Rightarrow \exists f \forall x_1 \cdots \forall x_n (f(\vec{x}) = f(\vec{x})) \\
(\exists_{R,n}) & \Rightarrow \exists R \forall x_1 \cdots \forall x_n (R(\vec{x}) \to R(\vec{x}))
\end{array}$$

Non-logical sequents

$$(\mathrm{Th}) \;\Rightarrow A \quad \text{for } A \in T$$

Figure 2.4. Simply T-closed sequents in $\mathcal{L}[\mathcal{C}]$ for a theory T in $\mathcal{L}$.

2.3.7 Proofs. Let T be a theory in a language $\mathcal{L}$, and p a valuation sequence in $\mathcal{L}$. We say that $\mathcal{D}$ *is proof of a valuation sequence* p *in* T, and write $\mathcal{D} \vdash_T p$, if $\mathcal{D}$ is a valuation tree for p, and the sequence $p +\!\!+ q$ is $T, \mathcal{L}$-closed for each leaf $q \in \mathcal{D}$. If necessary to avoid confusion, we will indicate the common language of T and p by writing $\mathcal{D} \vdash_T^{\mathcal{L}} p$.

We say that $\mathcal{D}$ *is a proof of a sequent* $\Gamma \Rightarrow \Delta$ *in* T, and write $\mathcal{D} \vdash_T \Gamma \Rightarrow \Delta$, if $\mathcal{D} \vdash_T p$ for some valuation sequence p such that $\Gamma \Rightarrow \Delta = \Gamma_p \Rightarrow \Delta_p$. We say that $\Gamma \Rightarrow \Delta$ *is provable in* T, and write $\vdash_T \Gamma \Rightarrow \Delta$, if there is a proof of $\Gamma \Rightarrow \Delta$ from T. We say that a sentence A of $\mathcal{L}$ is a *theorem in* T if $\vdash_T \Rightarrow A$.

We will abbreviate $(\mathcal{D}) \vdash_\emptyset \Gamma \Rightarrow \Delta$ to $(\mathcal{D}) \vdash \Gamma \Rightarrow \Delta$, and $(\mathcal{D}) \vdash_T \Rightarrow A$ to $(\mathcal{D}) \vdash_T A$. For the sake of readability, we will occasionally write $\Gamma \Rightarrow \Delta \;{}_T\!\dashv \mathcal{D}$ ($\Gamma \Rightarrow \Delta$ is proved by $\mathcal{D}$ in T), instead of $\mathcal{D} \vdash_T \Gamma \Rightarrow \Delta$.

2.3.8 Theorem (Soundness). *If* $\mathcal{D} \vdash_T \Gamma \Rightarrow \Delta$*, then* $T \vDash \Gamma \Rightarrow \Delta$.

Proof. Take any language $\mathcal{L}$ and any theory T in $\mathcal{L}$. We prove by induction on the cardinality of $\mathcal{D}$ for all p and $\mathcal{C}$ that

$$\textit{If } p \textit{ is a valuation sequence in } \mathcal{L}[\mathcal{C}] \textit{ and } \mathcal{D} \vdash_T^{\mathcal{L}[\mathcal{C}]} p, \textit{ then } T \vDash \Gamma_p \Rightarrow \Delta_p.$$

The soundness theorem will then follow immediately. Note that T is a theory in any expansion $\mathcal{L}[\mathcal{C}]$ of $\mathcal{L}$, and since p is in $\mathcal{L}[\mathcal{C}]$, $T \vDash \Gamma_p \Rightarrow \Delta_p$ means: each structure for $\mathcal{L}[\mathcal{C}]$ satisfying T satisfies also $\Gamma_p \Rightarrow \Delta_p$.

So take any $\mathcal{D}$ in any expansion $\mathcal{L}[\mathcal{C}][\mathcal{C}']$ of $\mathcal{L}[\mathcal{C}]$, and assume the induction hypothesis for all trees of smaller cardinality. Take any p and $\mathcal{C}$, assume $\mathcal{D} \vdash_T^{\mathcal{L}[\mathcal{C}]} p$, and take any structure $\mathcal{M}$ for $\mathcal{L}[\mathcal{C}]$ satisfying T. We have to show $\mathcal{M} \vDash \Gamma_p \Rightarrow \Delta_p$.

If $\mathcal{D} = \circ$, then p is $T, \mathcal{L}[\mathcal{C}]$-closed. If it is closed by 2.3.6(i), then $\mathcal{M} \vDash \Gamma_p \Rightarrow \Delta_p$ follows from the note in Par. 2.3.3 and satisfaction of all simply closed sequents in $\mathcal{M}$, which is easily proved from the definition of the satisfaction relation and basic properties of equality. Since all constants in p are already in the language $\mathcal{L}[\mathcal{C}]$, p can only be closed by 2.3.6(ii) if $p = q +\!\!+ \langle A[X/\mathsf{C}]*\rangle$ and $\exists X A \in \Gamma_q$ for some q and A with no occurrence of X. If $\mathcal{M} \vDash A$, then obviously $\mathcal{M} \vDash \exists X A, \Gamma_q \Rightarrow A, \Delta_q$. Otherwise $\mathcal{M} \nvDash \exists X A$, hence again $\mathcal{M} \vDash \exists X A, \Gamma_q \Rightarrow A, \Delta_q$. The situation is similar in case p is closed by 2.3.6(iii).

Otherwise, $\mathcal{D} = \begin{array}{c} A \\ \downarrow \\ \mathcal{D}_1 \end{array} \!\!\diagdown\, \mathcal{D}_2$ for some $\mathcal{D}_1$, $\mathcal{D}_2$, and a sentence A in $\mathcal{L}[\mathcal{C}][\mathcal{C}']$. As noted in Par. 2.3.4, at most one symbol from $\mathcal{C}'$ can occur in A due to regularity of $p +\!\!+ \langle A\rangle$ for $\mathcal{C}'$.

Assume first that A contains no symbol from $\mathcal{C}'$. For each node $q \in \mathcal{D}_1$, $(p +\!\!+ \langle A\rangle) +\!\!+ q$ is regular for $\mathcal{C}'$ since $\langle A\rangle +\!\!+ q$ is a node in $\mathcal{D}$. For each leaf $q \in \mathcal{D}_1$, $(p +\!\!+ \langle A\rangle) +\!\!+ q$ is $T, \mathcal{L}[\mathcal{C}]$-closed since $\langle A\rangle +\!\!+ q$ is a leaf in $\mathcal{D}$. Therefore $\mathcal{D}_1 \vdash_T^{\mathcal{L}[\mathcal{C}]} p +\!\!+ \langle A\rangle$. Because the cardinality of $\mathcal{D}_1$ is smaller than the cardinality of $\mathcal{D}$, we obtain $T \vDash A, \Gamma \Rightarrow \Delta$ from the induction hypothesis. Similarly, we have $\mathcal{D}_2 \vdash_T^{\mathcal{L}[\mathcal{C}]} p +\!\!+ \langle A*\rangle$, and obtain $T \vDash \Gamma \Rightarrow A, \Delta$ from IH. Therefore, we have $\mathcal{M} \vDash \Gamma \Rightarrow \Delta$ from the IH for $\mathcal{D}_1$ if $\mathcal{M} \vDash A$, and from the IH for $\mathcal{D}_2$ if $\mathcal{M} \nvDash A$.

Now, assume that A contains a symbol $\mathsf{C} \in \mathcal{C}'$. For each node $q \in \mathcal{D}_1$, $(p +\!\!+ \langle A\rangle) +\!\!+ q$ is regular for $\mathcal{C}'$, and therefore also regular for $\mathcal{C}' \setminus \{\mathsf{C}\}$. For each leaf $q \in \mathcal{D}_1$, since A is an unsigned sentence and the first occurrence of C, $(p +\!\!+ \langle A\rangle) +\!\!+ q$ cannot be $T, \mathcal{C}'$-closed by 2.3.6(ii) or (iii) for C. It must therefore be $T, \mathcal{L}[\mathcal{C}, \mathsf{C}]$-closed. Thus $\mathcal{D}_1 \vdash_T^{\mathcal{L}[\mathcal{C},\mathsf{C}]} p +\!\!+ \langle A\rangle$, and we obtain $T \vDash A, \Gamma_p \Rightarrow \Delta_p$ from the IH as above.

Since $p +\!\!+ \langle A\rangle$ is regular for $\mathcal{C}'$ and C cannot occur in p, there is B such that $A \equiv B[X/\mathsf{C}]$ and $\exists X B \in \Gamma_p$ or $A \equiv \neg B[X/\mathsf{C}]$ and $\forall X B \in \Delta_p$. In the first case, if the structure $\mathcal{M}$ does not satisfy $\exists X B$, we immediatelly have $\mathcal{M} \vDash \Gamma_p \Rightarrow \Delta_p$. Otherwise, there is an object m from $\mathcal{M}$ witnessing $\exists X B$, i.e., such that $\mathcal{M}[\mathsf{C} \mapsto m] \vDash B[X/\mathsf{C}]$. Therefore $\mathcal{M}[\mathsf{C} \mapsto m] \vDash \Gamma_p \Rightarrow \Delta_p$ by the induction hypothesis. Since C does not occur in $\Gamma_p \Rightarrow \Delta_p$, we also have $\mathcal{M} \vDash \Gamma_p \Rightarrow \Delta_p$. The proof of the second case is symmetric, but the object m is the counterexample of $\forall X B$ if $\mathcal{M} \nvDash \forall X B$. Note that in either case, the subtree $\mathcal{D}_2$ is irrelevant. □

Completeness

2.3.9 Towards completeness. We prove completeness of our proof calculus by the simplest possible method. Given a sequent $\Gamma \Rightarrow \Delta$, we construct a sequence of

valuation trees $\{\mathcal{D}_i\}_{i=1}^{\infty}$ for the sequent starting with $\mathcal{D}_0 \equiv \circ$. We obtain the tree $\mathcal{D}_{i+1}$ from $\mathcal{D}_i$ by replacing every open leaf with a new internal node $\begin{array}{c} A_i \\ \downarrow \\ \circ \end{array}\!\!\nwarrow\ \circ$. With a suitable enumeration of sentences $\{A_i\}_{i=1}^{\infty}$, we either stumble by this, rather brute-force, method at a valuation tree $\mathcal{D}_i$ proving $\Gamma \Rightarrow \Delta$ in T, or else we find an infinite sequence $\{q_i\}_{i=0}^{\infty}$ where each q_i is an open leaf in $\mathcal{D}_i$, and q_{i+1} is an immediate successor of q_i. This sequence defines two sets of sentences: $G := \bigcup_i \Gamma_{q_i}$ and $H := \bigcup_i \Delta_{q_i}$. We then construct a structure $\mathcal{M}$ from syntactic material which satisfies T, Γ, G, but does not satisfy any $B \in \Delta, H$, thus $\mathcal{M} \nvDash \Gamma \Rightarrow \Delta$.

2.3.10 Witnessing expansion. Let $\mathcal{L}$ be a language. Let $\mathcal{C}_0 := \emptyset$. For each i, let

$$\mathcal{C}_{i+1} := \mathcal{C}_i \cup \{\, \mathsf{C}_{\exists X A} \mid \exists X A \text{ is a sentence of } \mathcal{L}[\mathcal{C}_i] \text{ but not of } \mathcal{L}[\mathcal{C}_{i-1}] \text{ if } i > 0, \text{ and } \mathsf{C}_{\exists X A} \notin \mathcal{L}[\mathcal{C}_i] \,\}.$$

Let $\mathcal{C} := \bigcup_{i=0}^{\infty} \mathcal{C}_i$. We call the language $\mathcal{L}[\mathcal{C}]$ a *witnessing expansion of* $\mathcal{L}$, the constant $\mathsf{C}_{\exists X A}$ the *witnessing constant of* $\exists X A$, and the constant $\mathsf{C}_{\exists X \neg A}$ the *counterexample constant of* $\forall X A$.

2.3.11 Hintikka sets. The notation $\Gamma \Rightarrow \Delta$ was defined in Par. 2.3.3 for finite sets of sentences Γ, Δ only. Until the end of this section, we will use the notation $G \Rightarrow H$ also for possibly infinite sets of sentences G, H to denote possibly infinite sets of signed sentences. We define $\mathcal{M} \vDash G \Rightarrow H$ and $T \vDash G \Rightarrow H$ just as their finite counterparts.

A set of signed sentences $G \Rightarrow H$ is *simply consistent* if for each sentence $A \in G, H$ we have $A \in G$ iff $A \notin H$. A simply consistent set $G \Rightarrow H$ is *maximal* in a language $\mathcal{L}$ if it has no proper simply consistent superset in $\mathcal{L}$, i.e., for all sentences A of $\mathcal{L}$ either $A \in G$ or $A \in H$.

Let $\mathcal{L}[\mathcal{C}]$ be a witnessing expansion of $\mathcal{L}$, and $G \Rightarrow H$ a set of signed sentences of $\mathcal{L}[\mathcal{C}]$. We define the set of witnessing and counterexample constants for $G \Rightarrow H$

$$\mathcal{C}_{G \Rightarrow H} := \{\, \mathsf{C}_{\exists X A} \mid \exists X A \in G \,\} \cup \{\, \mathsf{C}_{\exists X \neg A} \mid \forall X A \in H \,\}.$$

The set $G \Rightarrow H$ is a *Hintikka set* if it is in the language $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$, and satisfies:

(i) $(t = t) \in G$ for all closed first-order terms t of $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$;

(ii) if $(s = t) \in G$ and $P[x/s] \in G$ for an atomic formula P and first-order terms t, s of $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$, then $P[x/t] \in G$;

(iii) if $(\neg A) \in G$, then $A \in H$;
if $(\neg A) \in H$, then $A \in G$;
if $(A \wedge B) \in G$, then $A \in G$ and $B \in G$;
if $(A \vee B) \in H$, then $A \in H$ and $B \in H$;
if $(A \to B) \in H$, then $A \in G$ and $B \in H$;

(iv) if $(A \wedge B) \in H$, then $A \in H$ or $B \in H$;
if $(A \vee B) \in G$, then $A \in G$ or $B \in G$;
if $(A \to B) \in G$, then $A \in H$ or $B \in G$;

(v) if $\exists X A \in H$, then $A[X/t] \in H$, and if $\forall X A \in G$, then $A[X/t] \in G$ for all t of the same sort as X in $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$;

(vi) $\exists x(x = x) \in G$ for some first-order variable x, and $\exists f \forall \vec{x}(f(\vec{x}) = f(\vec{x}))$, $\exists R \forall \vec{x}(R(\vec{x}) \to R(\vec{x})) \in G$ for each $n > 0$ and some n-ary function variable f, n-ary predicate variable R, and mutually distinct first-order variables $x_1, \ldots, x_n$;

(vii) if $\exists X A \in G$, then $A[X/\mathsf{C}_{\exists X A}] \in G$, and if $\forall X A \in H$, then $\neg A[X/\mathsf{C}_{\exists X \neg A}] \in H$.

2.3.12 Lemma. *For every simply consistent Hintikka set $G \Rightarrow H$ in a witnessing expansion $\mathcal{L}[\mathcal{C}]$ of $\mathcal{L}$ which is maximal in $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$, there is a structure $\mathcal{M}$ for $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$ which falsifies $G \Rightarrow H$, i.e., $\mathcal{M} \vDash G$, and $\mathcal{M} \nvDash A$ for all $A \in H$.*

Proof. We build the structure $\mathcal{M}$ from syntactic material. Let $\mathcal{C}_1$ denote the set of all first-order constants in $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$.

Let $\sim$ be a relation on $\mathcal{C}_1$ such that $\mathsf{c} \sim \mathsf{d}$ iff $(\mathsf{c} = \mathsf{d}) \in G$, and observe that $\sim$ is an equivalence: It is reflexive, since $(\mathsf{c} = \mathsf{c}) \in G$ by the property 2.3.11(i) of Hintikka sets. It is symmetric, since if $(\mathsf{c} = \mathsf{d}) \in G$, then we have $(\mathsf{d} = \mathsf{c}) \in G$ by 2.3.11(ii) for $P := (x = \mathsf{c})$, $s := \mathsf{c}$, $t := \mathsf{d}$. It is transitive, since if $(\mathsf{c} = \mathsf{d}), (\mathsf{d} = \mathsf{e}) \in G$, then we have $(\mathsf{c} = \mathsf{e}) \in G$ by 2.3.11(ii) for $P := (\mathsf{c} = x)$, $s := \mathsf{d}$, $t := \mathsf{e}$.

Let $[\mathsf{c}]$ denote the equivalence class of $\sim$ containing $\mathsf{c} \in \mathcal{C}_1$, i.e., the set $\{\, \mathsf{d} \mid \mathsf{d} \in \mathcal{C}_1, \mathsf{c} \sim \mathsf{d} \,\}$. We define the first-order domain M of $\mathcal{M}$ as the partition of $\mathcal{C}_1$ by the equivalence $\sim$, i.e.,

$$M := \{\, [\mathsf{c}] \mid \mathsf{c} \in \mathcal{C}_1 \,\}.$$

Note that M is non-empty, since we have the existential sentence $\exists x(x = x) \in G$ by 2.3.11(vi), and thus its witnessing constant $\mathsf{c}_{\exists x(x=x)} \in \mathcal{C}_1$. The meaning of each first-order constant c of $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$ in the structure $\mathcal{M}$ is defined as $\mathsf{c}^{\mathcal{M}} := [\mathsf{c}]$.

With the first-order domain fixed, we can assign meaning to second-order constants of $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$. Take any n-ary predicate constant R from $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$. If $(\mathsf{c}_1 = \mathsf{d}_1), \ldots, (\mathsf{c}_n = \mathsf{d}_n) \in G$ and $\mathsf{R}(\vec{\mathsf{c}}) \in G$, then by repeated application of the property 2.3.11(ii), we obtain $\mathsf{R}(\vec{\mathsf{d}}) \in G$. Therefore, we are allowed to define

$$([\mathsf{c}_1], \ldots, [\mathsf{c}_n]) \in \mathsf{R}^{\mathcal{M}} \text{ iff } \mathsf{R}(\vec{\mathsf{c}}) \in G.$$

Take any n-ary function constant f from $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$. We would like to define

$$\mathsf{f}^{\mathcal{M}}([\mathsf{c}_1], \ldots, [\mathsf{c}_n]) = [\mathsf{c}] \text{ iff } (\mathsf{f}(\vec{\mathsf{c}}) = \mathsf{c}) \in G.$$

This definition is valid only if there is always exactly one such class $[\mathsf{c}]$. Take any $[\mathsf{c}_1], \ldots, [\mathsf{c}_n] \in M$. The sentence $\exists x(\mathsf{f}(\vec{\mathsf{c}}) = x)$ of $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$ is in G, H by maximality. Observe that $\exists x(\mathsf{f}(\vec{\mathsf{c}}) = x) \notin H$, since then we would have $(\mathsf{f}(\vec{\mathsf{c}}) = \mathsf{f}(\vec{\mathsf{c}}))$ both in H by 2.3.11(v) and in G by 2.3.11(i), which would contradict the simple consistency of $G \Rightarrow H$. Thus $\exists x(\mathsf{f}(\vec{\mathsf{c}}) = x) \in G$. There is a witnessing constant $\mathsf{c}_{\exists x(\mathsf{f}(\vec{\mathsf{c}})=x)} \in \mathcal{C}_1$, and we have $(\mathsf{f}(\vec{\mathsf{c}}) = \mathsf{c}_{\exists x(\mathsf{f}(\vec{\mathsf{c}})=x)}) \in G$ by 2.3.11(vii). The equivalence class $[\mathsf{c}_{\exists x(\mathsf{f}(\vec{\mathsf{c}})=x)}]$ is the value of $\mathsf{f}^{\mathcal{M}}([\mathsf{c}_1], \ldots, [\mathsf{c}_n])$ provided it is unique. So take any $\mathsf{d}_1, \ldots, \mathsf{d}_n$ such that $[\mathsf{d}_i] = [\mathsf{c}_i]$, and assume there is e such that $(\mathsf{f}(\vec{\mathsf{d}}) = \mathsf{e}) \in G$. From $[\mathsf{d}_i] = [\mathsf{c}_i]$, we have

$(\mathsf{c}_i = \mathsf{d}_i) \in G$, and from suitable instances of 2.3.11(ii), we obtain $(\mathsf{f}(\vec{\mathsf{c}}) = \mathsf{f}(\vec{\mathsf{d}})) \in G$, and $(\mathsf{c}_{\exists x(\mathsf{f}(\vec{\mathsf{c}})=x)} = \mathsf{e}) \in G$. Thus $[\mathsf{c}_{\exists x(\mathsf{f}(\vec{\mathsf{c}})=x)}] = [\mathsf{e}]$, which proves uniqueness.

We complete the construction of $\mathcal{M}$ by defining for all $n > 0$ the sets

$$\mathcal{F}_n := \{\ \mathsf{f}^{\mathcal{M}} \mid \mathsf{f} \text{ is an } n\text{-ary function constant of } \mathcal{L}[\mathcal{C}_{G\Rightarrow H}]\ \},$$
$$\mathcal{R}_n := \{\ \mathsf{R}^{\mathcal{M}} \mid \mathsf{R} \text{ is an } n\text{-ary predicate constant of } \mathcal{L}[\mathcal{C}_{G\Rightarrow H}]\ \}.$$

For each $n > 0$, we have existential sentences $\exists f \forall x_1 \cdots \forall x_n (f(\vec{x}) = f(\vec{x}))$ and $\exists R \forall x_1 \cdots \forall x_n (R(\vec{x}) \to R(\vec{x}))$ in G by 2.3.11(vi), and hence their respective witnessing constants in $\mathcal{L}[\mathcal{C}_{G\Rightarrow H}]$. The sets $\mathcal{F}_n$ and $\mathcal{R}_n$ are therefore non-empty.

Note that for each object m of any sort from $\mathcal{M}$ there is a constant C of the same sort in $\mathcal{L}[\mathcal{C}_{G\Rightarrow H}]$ such that $\mathsf{C}^{\mathcal{M}} = m$. For $m \in M$, there is a first-order constant c of $\mathcal{L}[\mathcal{C}_{G\Rightarrow H}]$ such that $m = [\mathsf{c}] = \mathsf{c}^{\mathcal{M}}$. For $m \in \mathcal{F}_n$ (and similarly for $m \in \mathcal{R}_n$), there is a n-ary function constant f of $\mathcal{L}[\mathcal{C}_{G\Rightarrow H}]$ such that $\mathsf{f}^{\mathcal{M}} = m$ by the definition of $\mathcal{F}_n$.

Also note that we can prove by induction on construction using 2.3.11(ii) that for each first-order term t and each $\mathsf{c} \in \mathcal{C}_1$, we have

$$t^{\mathcal{M}} = [\mathsf{c}] \text{ iff } (t = \mathsf{c}) \in G. \tag{$\dagger$}$$

We are now ready to prove by induction on the size (number of symbols) of sentences A of $\mathcal{L}[\mathcal{C}_{G\Rightarrow H}]$ that $\mathcal{M} \vDash A$ iff $A \in G$. We will then have $\mathcal{M} \vDash G$ and $\mathcal{M} \nvDash A$ for all $A \in H$, so $\mathcal{M}$ will falsify $G \Rightarrow H$.

Take any sentence A of $\mathcal{L}[\mathcal{C}_{G\Rightarrow H}]$. Assume first that A is atomic. If A is an equality $t = s$ of first-order terms, there are $[\mathsf{c}], [\mathsf{d}] \in M$ such that $t^{\mathcal{M}} = [\mathsf{c}]$ and $s^{\mathcal{M}} = [\mathsf{d}]$, hence $(t = \mathsf{c}), (s = \mathsf{d}) \in G$ by ($\dagger$). If $(t = s) \in G$, then we obtain $(\mathsf{c} = \mathsf{d}) \in G$ from 2.3.11(ii), therefore $t^{\mathcal{M}} = [\mathsf{c}] = [\mathsf{d}] = s^{\mathcal{M}}$, hence $\mathcal{M} \vDash t = s$. In the opposite direction, if $\mathcal{M} \vDash t = s$, we have $[\mathsf{c}] = t^{\mathcal{M}} = s^{\mathcal{M}} = [\mathsf{d}]$, hence $(\mathsf{c} = \mathsf{d}) \in G$ by the definition of $\sim$, and consequently $(t = s) \in G$ by 2.3.11(ii).

If A is a predicate application $\mathsf{R}(t_1, \ldots, t_n)$, there are $[\mathsf{c}_i] \in M$ such that $t_i^{\mathcal{M}} = [\mathsf{c}_i]$, hence $(t_i = \mathsf{c}_i) \in G$ by ($\dagger$) for all i, $1 \leq i \leq n$. If $\mathsf{R}(\vec{t}) \in G$, then we obtain $\mathsf{R}(\vec{\mathsf{c}}) \in G$ from 2.3.11(ii), hence $\mathcal{M} \vDash \mathsf{R}(\vec{t})$. If $\mathcal{M} \vDash \mathsf{R}(\vec{t})$, we have $([\mathsf{c}_1], \ldots, [\mathsf{c}_n]) \in \mathsf{R}^{\mathcal{M}}$, hence $\mathsf{R}(\vec{\mathsf{c}}) \in G$ by the definition of $\sim$, and consequently $\mathsf{R}(\vec{t}) \in G$ by 2.3.11(ii).

Assume A is constructed by a propositional connective. The proof is similar for all connectives, so take, for instance, the case $A \equiv B \to C$. If $A \in G$, we have $B \in H$ or $C \in G$ by 2.3.11(iv). In the first case, we have $\mathcal{M} \nvDash B$ by the induction hypothesis, and consequently $\mathcal{M} \vDash A$. In the second case, $\mathcal{M} \vDash C$ by the IH, and $\mathcal{M} \vDash A$ again. In the opposite direction, if $A \notin G$, then $A \in H$ by maximality, and hence $B \in G$ and $C \in H$ by 2.3.11(iii). We thus have $\mathcal{M} \vDash B$ and $\mathcal{M} \nvDash C$ by the IH, so $\mathcal{M} \nvDash A$.

Assume $A \equiv \exists X B$. If $A \in G$, then $B[X/\mathsf{C}_A] \in G$ by 2.3.11(vii), so $\mathcal{M} \vDash B[X/\mathsf{C}_A]$ by the IH. Therefore, for a constant C of the sort of X not in $\mathcal{L}[\mathcal{C}_{G\Rightarrow H}]$, we have $\mathcal{M}[\mathsf{C} \mapsto \mathsf{C}_A^{\mathcal{M}}] \vDash B[X/\mathsf{C}]$, and consequently $\mathcal{M} \vDash A$. In the opposite direction, assume $\mathcal{M} \vDash A$. There is an object m of the sort of X from $\mathcal{M}$, such that for any constant C of the same sort not in $\mathcal{L}[\mathcal{C}_{G\Rightarrow H}]$ we have $\mathcal{M}[\mathsf{C} \mapsto m] \vDash B[X/\mathsf{C}]$. Since $m = \mathsf{D}^{\mathcal{M}}$ for some constant D in $\mathcal{L}[\mathcal{C}_{G\Rightarrow H}]$, we also have $\mathcal{M} \vDash B[X/\mathsf{D}]$. If it were the case that

$A \in H$, then we would have $B[X/\mathsf{D}] \in H$ by 2.3.11(v), and hence a contradiction $\mathcal{M} \nvDash B[X/\mathsf{D}]$ by the IH. Therefore $A \notin H$, so $A \in G$ by maximality.

The case when $A \equiv \forall X B$ is similar. □

2.3.13 Brute-force proof construction. Let T be a theory in a language $\mathcal{L}$, p a valuation sequence in $\mathcal{L}$, $\mathcal{L}[\mathcal{C}]$ a witnessing expansion of $\mathcal{L}$, and $\{A_i\}_{i=0}^{\infty}$ an enumeration of all sentences in $\mathcal{L}[\mathcal{C}]$ without repetition. We define an infinite sequence $\{\mathcal{D}_i\}_{i=0}^{\infty}$ of valuation trees for p as follows:

$$\mathcal{D}_0 = \circ$$
$$\mathcal{D}_{i+1} = \mathcal{D}_i \cup \{\, q \mathbin{+\!\!+} \langle A_q \rangle, q \mathbin{+\!\!+} \langle A_q * \rangle \mid q \in \mathcal{D}_i \text{ is a leaf which is not } T\text{-closed} \,\}$$

where the sentence A_q is A_j for the least j such that $A_j \notin \Gamma_q, \Delta_q$, and $p \mathbin{+\!\!+} q \mathbin{+\!\!+} \langle A_j \rangle$ is *strongly regular for* $\mathcal{C}$, i.e., it is regular for $\mathcal{C}$, and if $A_j \equiv B[X/\mathsf{C}]$ and $\mathsf{C} \in \mathcal{C}$ does not occur in q, then $\mathsf{C} \equiv \mathsf{C}_{\exists X B}$.

2.3.14 Theorem (Completeness). *If $T \vDash \Gamma \Rightarrow \Delta$, then there is such a tree $\mathcal{D}$ that $\mathcal{D} \vdash_T \Gamma \Rightarrow \Delta$.*

Proof. Let $\mathcal{L}$ be the common language of T and $\Gamma \Rightarrow \Delta$, and assume $T \vDash \Gamma \Rightarrow \Delta$. Let $\mathcal{L}[\mathcal{C}]$ be a witnessing expansion of $\mathcal{L}$, and let $\mathcal{C}_i$ be defined as in Par. 2.3.10. Take any sequence p such that $(\Gamma \Rightarrow \Delta) = (\Gamma_p \Rightarrow \Delta_p)$, and let $\{A_i\}_{i=0}^{\infty}$ and $\{\mathcal{D}_i\}_{i=0}^{\infty}$ be the infinite sequences from Par. 2.3.13. If $\mathcal{D}_i = \mathcal{D}_{i+1}$ for some i, the tree $\mathcal{D}_i$ is a proof of $\Gamma \Rightarrow \Delta$ in T, and we are done.

If there is no such i, we get a contradiction by showing that $T \nvDash \Gamma \Rightarrow \Delta$. Let $\mathcal{D} := \bigcup_{i=0}^{\infty} \mathcal{D}_i$. $\mathcal{D}$ is an infinite set-theoretical tree. Since each its internal node has exactly two successors, $\mathcal{D}$ is finitely branching. By König's lemma, there is an infinite path $\{q_i\}_{i=0}^{\infty}$ through $\mathcal{D}$ where $q_0 = \langle\rangle$ and q_{i+1} is an immediate successor of q_i for each i. By induction on i, we can easily show that $q_i \in \mathcal{D}_i$, and that $q_{i+1} = q_i \mathbin{+\!\!+} \langle A_{q_i} \rangle$ or $q_{i+1} = q_i \mathbin{+\!\!+} \langle A_{q_i} * \rangle$.

Let $G := \bigcup_{i=0}^{\infty} \Gamma_{q_i}$ and $H := \bigcup_{i=0}^{\infty} \Delta_{q_i}$. We will show below that $G \Rightarrow H$ is (a) simply consistent, (b) maximal in $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$, (c) a superset of $\Gamma \Rightarrow \Delta$, (d) a superset of T, and (e) a Hintikka set. The structure $\mathcal{M}$ from Lemma 2.3.12 for $\mathcal{L}[\mathcal{C}_{G \Rightarrow H}]$ will then satisfy T, Γ and not satisfy any sentence in Δ. The reduct $\mathcal{M}'$ of $\mathcal{M}$ to $\mathcal{L}$ will thus witness $T \nvDash \Gamma \Rightarrow \Delta$.

(a) Assume $G \Rightarrow H$ is not simply consistent. Then, some sentence A is a member of both G and H. Therefore, there k, q, and r such that $A_{q_k} \equiv A$, and $q_k = q \mathbin{+\!\!+} \langle A \rangle \mathbin{+\!\!+} r$ or $q_k = q \mathbin{+\!\!+} \langle A * \rangle \mathbin{+\!\!+} r$. This contradicts the condition $A_{q_k} \notin \Gamma_{q_k}, \Delta_{q_k}$ from Par. 2.3.13.

(b) We show by induction on n that for all i, if A_i is a sentence of $\mathcal{L}[\mathcal{C}_{G \Rightarrow H} \cap \mathcal{C}_n]$, then $A_i \equiv A_{q_k}$ for some k. For the basis of the induction, take any A_i of $\mathcal{L}[\mathcal{C}_{G \Rightarrow H} \cap \mathcal{C}_0] = \mathcal{L}$. Take the least k such that $A_{q_k} \equiv A_j$ for some $j \geq i$. By the definition of A_{q_k}, j is the least number such that $p \mathbin{+\!\!+} q_k \mathbin{+\!\!+} \langle A_j \rangle$ is strongly regular for $\mathcal{C}$. The sequence $p \mathbin{+\!\!+} q_k \mathbin{+\!\!+} \langle A_i \rangle$ is trivially strongly regular, since A_i contains no symbol from $\mathcal{C}$. Therefore, we must have $j = i$.

For the inductive step, assume the induction hypothesis for n, take any sentence A_i of $\mathcal{L}[\mathcal{C}_{G\Rightarrow H} \cap \mathcal{C}_{n+1}]$. If A_i is a sentence of $\mathcal{L}[\mathcal{C}_{G\Rightarrow H} \cap \mathcal{C}_n]$, we obtain k directly from the induction hypothesis. Otherwise at least one constant from $\mathcal{C}_{n+1} \setminus \mathcal{C}_n$ occurs in A_i. Assume first, that $A_i \equiv B[X/\mathsf{C}_{\exists XB}]$ for some formula B of $\mathcal{L}[\mathcal{C}_{G\Rightarrow H} \cap \mathcal{C}_n]$. By the IH, there is k' such that $A_{q_{k'}} \equiv \exists XB$. Moreover, if $B \equiv \neg B'$, there is k'' such that $A_{q_{k''}} \equiv \forall XB'$ by the IH. By the definition of $\mathcal{C}_{G\Rightarrow H}$, we have $\exists XB \in G$ or $\forall XB' \in H$. If both cases apply, let $k_1 := \min\{k', k''\}$; if only $\exists XB \in G$, let $k_1 := k'$; if only $\forall XB' \in H$, let $k_1 := k''$. Take the least $k > k_1$ such that $A_{q_k} \equiv A_j$ for some $j \geq i$. The sentence A_i cannot occur in q_k, since the constant $\mathsf{C}_{\exists XB}$ cannot occur there. If it did, there would be a proper prefix q_ℓ of q_k, $\ell < k$, where $\mathsf{C}_{\exists XB}$ would occur in A_{q_ℓ} and not in q_ℓ. By strong regularity, we would have $A_{q_\ell} \equiv A_i$. But if $\ell \leq k_1$, then $p +\!\!+ q_\ell +\!\!+ \langle A\rangle$ would not be regular. Otherwise the existence of ℓ would contradict the minimality of k. Since $k_1 < k$ and by the definition of $\mathcal{C}_{G\Rightarrow H}$, we have $\exists XB \in \Gamma_{q_k}$ or $\forall XB' \in \Delta_{q_k}$. Therefore $p +\!\!+ q_k +\!\!+ \langle A_i\rangle$ is strongly regular, hence $i = j$.

Now assume that constants $\mathsf{C}_{\exists X_1 B_1}, \ldots, \mathsf{C}_{\exists X_m B_m} \in \mathcal{C}_{n+1} \setminus \mathcal{C}_n$ occur in A_i, and the above case does not apply. For each j, $1 \leq j \leq m$, there is k_j such that $A_{q_{k_j}} \equiv B_j[X_j/\mathsf{C}_{\exists X_j B_j}]$ as proved above. Let $k > \max\{k_j\}_{j=1}^m$ be the least such that $A_{q_k} \equiv A_j$ for $j \geq i$. If A_i occurred in q_k, there would be a proper prefix q_ℓ of q_k, $\ell < k$, with $A_{q_\ell} \equiv A_i$. But if $\ell < \max\{k_j\}_{j=1}^m$, then $p +\!\!+ q_\ell +\!\!+ \langle A_{q_\ell}\rangle$ would not be strongly regular, since A_i would precede some $B_j[X_j/\mathsf{C}_{\exists X_j B_j}]$. The case $\ell = \max\{k_j\}_{j=1}^m$ contradicts the assumption since A_i would be equal to some $B_j[X_j/\mathsf{C}_{\exists X_j B_j}]$. Finally, $\ell > \max\{k_j\}_{j=1}^m$, would contradict the minimality of k. Therefore, A_i does not occur in q_k. Since all constants $\mathsf{C}_{\exists X_j B_j}$ are already present in q_k by the choice of k, the sequence $p +\!\!+ q_k +\!\!+ \langle A_i\rangle$ is strongly regular, hence $i = j$.

(c) Take any $A \in T$. By maximality, we have $A \in G, H$. If it were the case that $A \in H$, there would be q_k such that $A \in \Delta_{q_k}$, so q_k would be T-closed by (Th) in Fig. 2.4, which is a contradiction. Therefore $A \in G$.

(d) Take any $A \in \Gamma$. By maximality, we have $A \in G, H$. If it were the case that $A \in H$, there would be q_k such that $A \in \Delta_{q_k}$, but then $p +\!\!+ q_k$ would be T-closed by (Ax) in Fig. 2.4, which is a contradiction. Therefore $A \in G$. Similarly, we can prove that any $A \in \Delta$ is in H.

(e) It remains to prove that $G \Rightarrow H$ is a Hintikka set. We must first show that $G \Rightarrow H$ is in $\mathcal{L}[\mathcal{C}_{G\Rightarrow H}]$, i.e., all constants occurring in $G \Rightarrow H$ are of $\mathcal{L}[\mathcal{C}_{G\Rightarrow H}]$. Take any constant $\mathsf{C} \in \mathcal{C}$ occurring in $G \Rightarrow H$. By the definition of $G \Rightarrow H$, there is k such that C does not occur in q_k, and occurs in A_{q_k}. By strong regularity, $\mathsf{C} \equiv \mathsf{C}_{\exists XB}$ for some formula B such that $A_{q_k} \equiv B[X/\mathsf{C}]$. By regularity, there is $\exists XB \in \Gamma_{q_k} \subseteq G$ or $\forall XB \in \Delta_{q_k} \subseteq H$. Therefore $\mathsf{C} \in \mathcal{C}_{G\Rightarrow H}$.

For the condition (i) from Par. 2.3.11, we have $(t = t) \in G, H$ by maximality, and $(t = t) \in G$ since otherwise some q_k would be T-closed by (Refl) in Fig. 2.4. A similar argument applies to conditions (ii) to (vi).

The condition (vii) needs special treatment: Assume $\exists XA \in G$. By maximality, and since $\mathsf{C}_{\exists XA} \in \mathcal{C}_{G\Rightarrow H}$, we have $A[X/\mathsf{C}_{\exists XA}] \in G, H$. By the definition of $G \Rightarrow H$, there is k such that $\mathsf{C}_{\exists XA}$ does not occur in q_k, and occurs in A_{q_k}, and by strong regularity, $A_{q_k} \equiv A[X/\mathsf{C}_{\exists XA}]$, and $\exists XA \in \Gamma_p, \Gamma_{q_k}$. Therefore, the sequence

$p +\!\!+ q_k +\!\!+ \langle A[X/\mathsf{C}_{\exists XA}]* \rangle$ is T-closed by Par. 2.3.6(ii), so we must have $q_{k+1} = q_k +\!\!+ \langle A[X/\mathsf{C}_{\exists XA}] \rangle$, and consequently $A[X/\mathsf{C}_{\exists XA}] \in G$. □

2.4 Relationship with Sequent Calculus and Tableaux

The formal system introduced in the previous section is closely related to sequent calculus (originally due to Gentzen; many variants exist, extensively surveyed in, e.g., [TS00]) and tableaux (originally due to Beth, but modified and popularized by Smullyan, [Smu68]). In both of these systems, formal proofs are trees built by rules of inference. In sequent calculus, nodes of the proof tree are labeled with sequents; in tableaux, nodes are labeled with signed formulas.

Our formal system with valuation trees uses case analysis on the truth value of a formula as its only rule. Sequent calculus and tableaux contain an equivalent rule, called the cut. However, they also contain many other rules for elimination of propositional connectives and quantifiers from assumptions and from goals. In our system, these rules can be simulated by case analysis and axioms. On the other hand, axioms of our system can be easily proved in sequent calculus as well as in tableaux. Overall, proofs in one of the three systems (our system, sequent calculus, and tableaux) can be quite straightforwardly translated into the other two systems.

2.4.1 Example. We will illustrate the relationship between our formal system, sequent calculus, and tableaux by means of an example. We will compare formal proofs of the distribution of the general quantifier over implication:

$$D :\equiv \forall x(\mathsf{Q}(x) \to \mathsf{R}(x)) \to \forall x\mathsf{Q}(x) \to \forall x\mathsf{R}(x).$$

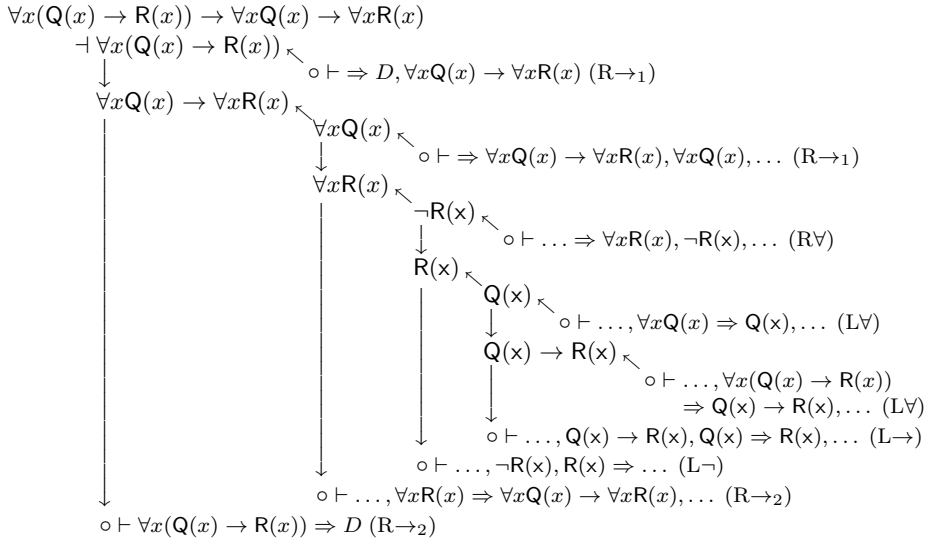
Informally, the proof is conducted as follows: Assume $\forall x(\mathsf{Q}(x) \to \mathsf{R}(x))$ and $\forall x\mathsf{Q}(x)$, and take any x. If we prove $\mathsf{R}(\mathsf{x})$, we can conclude $\forall x\mathsf{R}(x)$, since x is new and therefore arbitrary. By the first assumption, we have $\mathsf{Q}(\mathsf{x}) \to \mathsf{R}(\mathsf{x})$, and since the antecedent $\mathsf{Q}(\mathsf{x})$ holds by the second assumption, we conclude $\mathsf{R}(\mathsf{x})$.

The formalizations of the above proof are depicted in Fig. 2.5. Part (a) shows the valuation tree proving D, in which we have annotated all leaves with their respective simply closed sequents. Note that these annotations are not a part of the valuation tree.

Part (b) of Fig. 2.5 depicts Smullyan's tableau proof of D ([Smu68, §V.2]). The tableau starts with assuming that D is false (1), which is indicated by prefixing it with **F**. From this it follows that the antecedent of D is true ((2), indicated by **T**), and the consequent is false (3). Further consequences are added. Sometimes, it is necessary to split the proof into separate branches. In particular, from the assumption (7) $\mathbf{T}\,\mathsf{Q}(\mathsf{x}) \to \mathsf{R}(\mathsf{x})$, we can infer that either the antecedent is false (9), or the consequent is true (10). The tableau proof is complete when a contradiction (indicated by □) is found in each branch.

Note that there is a simple connection between the tableau and the valuation tree: Whenever a sentence is assumed true in the tableau, the right child of its node

(a)



(b)

(1)	$\mathbf{F}\,\forall x(\mathsf{Q}(x) \to \mathsf{R}(x)) \to \forall x\mathsf{Q}(x) \to \forall x\mathsf{R}(x)$		
(2)	$\mathbf{T}\,\forall x(\mathsf{Q}(x) \to \mathsf{R}(x))$		(1)
(3)	$\mathbf{F}\,\forall x\mathsf{Q}(x) \to \forall x\mathsf{R}(x)$		(1)
(4)	$\mathbf{T}\,\forall x\mathsf{Q}(x)$		(3)
(5)	$\mathbf{F}\,\forall x\mathsf{R}(x)$		(3)
(6)	$\mathbf{F}\,\mathsf{R}(\mathsf{x})$		(5)
(7)	$\mathbf{T}\,\mathsf{Q}(\mathsf{x}) \to \mathsf{R}(\mathsf{x})$		(2)
(8)	$\mathbf{T}\,\mathsf{Q}(\mathsf{x})$		(4)
(9)	$\mathbf{F}\,\mathsf{Q}(\mathsf{x})$	(10) $\mathbf{T}\,\mathsf{R}(\mathsf{x})$	(7)
	$\square$ (8, 9)		$\square$ (6, 10)

(c)

$$\mathrm{L}\to\ \frac{\ldots, \mathbf{Q}(\boldsymbol{x}) \Rightarrow \mathbf{Q}(\boldsymbol{x}), \mathsf{R}(x) \qquad \ldots, \mathbf{R}(\boldsymbol{x}) \Rightarrow \mathbf{R}(\boldsymbol{x})}{\mathrm{L}\forall\ \dfrac{\ldots, \mathbf{Q}(\boldsymbol{x}) \to \mathbf{R}(\boldsymbol{x}), \mathsf{Q}(x) \Rightarrow \mathsf{R}(x)}{\mathrm{L}\forall\ \dfrac{\ldots, \forall \boldsymbol{x}\mathbf{Q}(\boldsymbol{x}), \mathsf{Q}(x) \to \mathsf{R}(x) \Rightarrow \mathsf{R}(x)}{\mathrm{R}\forall\ \dfrac{\forall \boldsymbol{x}(\mathbf{Q}(\boldsymbol{x}) \to \mathbf{R}(\boldsymbol{x})), \forall x\mathsf{Q}(x) \Rightarrow \mathsf{R}(x)}{\mathrm{R}\to\ \dfrac{\forall x(\mathsf{Q}(x) \to \mathsf{R}(x)), \forall x\mathsf{Q}(x) \Rightarrow \forall \boldsymbol{x}\mathbf{R}(\boldsymbol{x})}{\mathrm{R}\to\ \dfrac{\forall x(\mathsf{Q}(x) \to \mathsf{R}(x)) \Rightarrow \forall \boldsymbol{x}\mathbf{Q}(\boldsymbol{x}) \to \forall \boldsymbol{x}\mathbf{R}(\boldsymbol{x})}{\Rightarrow \forall \boldsymbol{x}(\mathbf{Q}(\boldsymbol{x}) \to \mathbf{R}(\boldsymbol{x})) \to \forall \boldsymbol{x}\mathbf{Q}(\boldsymbol{x}) \to \forall \boldsymbol{x}\mathbf{R}(\boldsymbol{x})}}}}}}$$

Figure 2.5. Comparison of proofs in (a) our valuation trees calculus, (b) Smullyan's semantic tableaux, (c) sequent calculus $\mathbf{G3c}^{=}$.

in the valuation tree is closed, and the rest of the tableau is mapped to the left branch of the valuation tree in a similar manner. The situation is reversed when a sentence is assumed false in the tableau. The tableau calculus allows for a simpler introduction of the counterexample constant x than our valuation tree formal system, which requires a detour through $\neg\mathsf{R}(\mathsf{x})$.

Part (c) of Fig. 2.5 depicts a derivation of the sequent $\Rightarrow D$ in the sequent calculus $\mathbf{G3c}^{=}$ from [TS00]. In contrast to both our system and tableaux, the proofs in sequent calculus are reversed, with the proved sequent at the bottom, and the axioms at the top. In the top three rows of the derivation, we have replaced with "..." formulas which are no longer needed.

An application of a sequent calculus rule often corresponds to two case analyses in the valuation tree (a) (and two lines in the tableau (b)). For instance, the bottom-most rule R$\to$ corresponds to the case analysis on the antecedent of D and the case analysis on the consequent of D in the valuation tree (a). A formula appears in the antecedent in the sequent proof iff the right child of its node in the valuation tree is closed.

2.4.2 Translation of proofs in the sequent calculus to valuation trees. A generalization of the example from the previous paragraph is possible. Each derivation $\mathcal{D}$ of a sequent $\Gamma \Rightarrow \Delta$ in the sequent calculus can be straightforwardly translated to a valuation tree $\mathcal{D}^*$ proving $\Gamma \Rightarrow \Delta$. This can be proved by induction on the construction of $\mathcal{D}$ with the translation given in Fig. 2.6 for propositional rules and in Fig. 2.7 for equational and quantifier rules of the sequent calculus $\mathbf{G3c}^{=}$ of [TS00, §§3.5, 4.7].

Let us, for instance, explain the translation of the rule L$\to$. A derivation $\mathcal{D}$ of a sequent which has L$\to$ as its last rule derives some sequent $A \to B, \Gamma \Rightarrow \Delta$. The rule L$\to$ splits the derivation to $\mathcal{D}_1$ deriving $\Gamma \Rightarrow A, \Delta$ and $\mathcal{D}_2$ deriving $B, \Gamma \Rightarrow \Delta$, i.e.,

$$\mathrm{L}\to\ \frac{\overset{\mathcal{D}_1}{\overline{\Gamma \Rightarrow A, \Delta}} \qquad \overset{\mathcal{D}_2}{\overline{B, \Gamma \Rightarrow \Delta}}}{A \to B, \Gamma \Rightarrow \Delta}\,.$$

Let us assume that we have obtained $\mathcal{D}_1^* \vdash B, \Gamma \Rightarrow \Delta$ and $\mathcal{D}_2^* \vdash \Gamma \Rightarrow A, \Delta$ by inductively applying the translation to $\mathcal{D}_1$ and $\mathcal{D}_2$. We then translate the derivation $\mathcal{D}$ to the following valuation tree $\mathcal{D}^*$ proving $A \to B, \Gamma \Rightarrow \Delta$:

$$\mathcal{D}^* = \begin{array}{l} A \\ \downarrow \\ B \\ \downarrow \\ \mathcal{D}_2^*\,. \end{array} \quad \begin{array}{l} \nwarrow \\ \mathcal{D}_1^* \\ \nwarrow \\ \circ \end{array}$$

For any path p whose sequent is $A \to B, \Gamma \Rightarrow \Delta$ and for the leaf node $q = \langle A, B* \rangle$ in $\mathcal{D}^*$, $p +\!\!+ q$ is closed because its sequent $A \to B, A, \Gamma \Rightarrow B, \Delta$ is a superset of the simply closed sequent (L$\to$) from Fig. 2.4.

Note that in quantifier rules L$\exists$ and R$\forall$, y is a new free variable, which must be replaced by a new constant c_y in the valuation tree.

	$\mathcal{D}$	$\mathcal{D}^*$
Ax	$A, \Gamma \Rightarrow A, \Delta$	$A, \Gamma \Rightarrow A, \Delta \dashv \circ\ (\mathrm{Ax})$
Cut	$\dfrac{\overset{\mathcal{D}_1}{\overline{A}, \overline{\Gamma} \Rightarrow \overline{\Delta}} \qquad \overset{\mathcal{D}_2}{\overline{\Gamma} \Rightarrow \overline{A}, \overline{\Delta}}}{\Gamma \Rightarrow \Delta}$	$\Gamma \Rightarrow \Delta \dashv A$ $\downarrow$ $\mathcal{D}_2^* \vdash \Gamma \Rightarrow A, \Delta$ $\mathcal{D}_1^* \vdash A, \Gamma \Rightarrow \Delta$
L¬	$\dfrac{\overset{\mathcal{D}_1}{\overline{\Gamma} \Rightarrow \overline{A}, \overline{\Delta}}}{\neg A, \Gamma \Rightarrow \Delta}$	$\neg A, \Gamma \Rightarrow \Delta \dashv A$ $\downarrow$ $\mathcal{D}_1^* \vdash \Gamma \Rightarrow A, \Delta$ $\circ \vdash \neg A, A, \Gamma \Rightarrow \Delta\ (\mathrm{L}\neg)$
R¬	$\dfrac{\overset{\mathcal{D}_1}{\overline{A}, \overline{\Gamma} \Rightarrow \overline{\Delta}}}{\Gamma \Rightarrow \neg A, \Delta}$	$\Gamma \Rightarrow \neg A, \Delta \dashv A$ $\downarrow$ $\circ \vdash \Gamma \Rightarrow \neg A, A, \Delta\ (\mathrm{R}\neg)$ $\mathcal{D}_1^* \vdash A, \Gamma \Rightarrow \Delta$
L∧	$\dfrac{\overset{\mathcal{D}_1}{\overline{A}, \overline{B}, \overline{\Gamma} \Rightarrow \overline{\Delta}}}{A \wedge B, \Gamma \Rightarrow \Delta}$	$A \wedge B, \Gamma \Rightarrow \Delta \dashv A$ $\downarrow$ $\circ \vdash A \wedge B, \Gamma \Rightarrow A, \Delta\ (\mathrm{L}\wedge_1)$ B $\downarrow$ $\circ \vdash A \wedge B, \Gamma \Rightarrow B, \Delta\ (\mathrm{L}\wedge_2)$ $\mathcal{D}_1^* \vdash A, B, \Gamma \Rightarrow \Delta$
R∧	$\dfrac{\overset{\mathcal{D}_1}{\overline{\Gamma} \Rightarrow \overline{A}, \overline{\Delta}} \qquad \overset{\mathcal{D}_2}{\overline{\Gamma} \Rightarrow \overline{B}, \overline{\Delta}}}{\Gamma \Rightarrow A \wedge B, \Delta}$	$\Gamma \Rightarrow A \wedge B, \Delta \dashv A$ $\downarrow$ $\mathcal{D}_1^* \vdash \Gamma \Rightarrow A, \Delta$ B $\downarrow$ $\mathcal{D}_2^* \vdash \Gamma \Rightarrow B, \Delta$ $\circ \vdash A, B, \Gamma \Rightarrow A \wedge B, \Delta\ (\mathrm{R}\wedge)$
L∨	$\dfrac{\overset{\mathcal{D}_1}{\overline{A}, \overline{\Gamma} \Rightarrow \overline{\Delta}} \qquad \overset{\mathcal{D}_2}{\overline{B}, \overline{\Gamma} \Rightarrow \overline{\Delta}}}{A \vee B, \Gamma \Rightarrow \Delta}$	$A \vee B, \Gamma \Rightarrow \Delta \dashv A$ $\downarrow$ B $\downarrow$ $\circ \vdash A \vee B, \Gamma \Rightarrow A, B, \Delta\ (\mathrm{L}\vee)$ $\downarrow$ $\mathcal{D}_2^* \vdash B, \Gamma \Rightarrow \Delta$ $\mathcal{D}_1^* \vdash A, \Gamma \Rightarrow \Delta$
R∨	$\dfrac{\overset{\mathcal{D}_1}{\overline{\Gamma} \Rightarrow \overline{A}, \overline{B}, \overline{\Delta}}}{\Gamma \Rightarrow A \vee B, \Delta}$	$\Gamma \Rightarrow A \vee B, \Delta \dashv A$ $\downarrow$ B $\downarrow$ $\mathcal{D}_1^* \vdash \Gamma \Rightarrow A, B, \Delta$ $\downarrow$ $\circ \vdash B, \Gamma \Rightarrow A \vee B, \Delta\ (\mathrm{R}\vee_2)$ $\circ \vdash A, \Gamma \Rightarrow A \vee B, \Delta\ (\mathrm{R}\vee_1)$
L→	$\dfrac{\overset{\mathcal{D}_1}{\overline{\Gamma} \Rightarrow \overline{A}, \overline{\Delta}} \qquad \overset{\mathcal{D}_2}{\overline{B}, \overline{\Gamma} \Rightarrow \overline{\Delta}}}{A \to B, \Gamma \Rightarrow \Delta}$	$A \to B, \Gamma \Rightarrow \Delta \dashv A$ $\downarrow$ $\mathcal{D}_1^* \vdash \Gamma \Rightarrow A, \Delta$ B $\downarrow$ $\circ \vdash A \to B, A, \Gamma \Rightarrow B, \Delta\ (\mathrm{L}\to)$ $\mathcal{D}_2^* \vdash B, \Gamma \Rightarrow \Delta$
R→	$\dfrac{\overset{\mathcal{D}_1}{\overline{A}, \overline{\Gamma} \Rightarrow \overline{B}, \overline{\Delta}}}{\Gamma \Rightarrow A \to B, \Delta}$	$\Gamma \Rightarrow A \to B, \Delta \dashv A$ $\downarrow$ $\circ \vdash \Gamma \Rightarrow A, A \to B, \Delta\ (\mathrm{R}\to_1)$ B $\downarrow$ $\mathcal{D}_1^* \vdash A, \Gamma \Rightarrow B, \Delta$ $\circ \vdash B, \Gamma \Rightarrow A \to B, \Delta\ (\mathrm{R}\to_2)$

Figure 2.6. Translation of **G3c**$^{=}$ proofs to valuation trees (propositional rules).

	$\mathcal{D}$	$\mathcal{D}^*$
Refl	$\dfrac{\overline{t = t}, \overline{\Gamma} \Rightarrow \overline{\Delta}}{\Gamma \Rightarrow \Delta}\ \mathcal{D}_1$	$\Gamma \Rightarrow \Delta \dashv t = t$ ↖ ∘ $\vdash \Gamma \Rightarrow t = t, \Delta$ (Refl) ↓ $\mathcal{D}_1^* \vdash t = t, \Gamma \Rightarrow \Delta$
Repl	$\dfrac{\overline{s = t}, \overline{P[x/s]}, \overline{P[x/t]}, \overline{\Gamma} \Rightarrow \overline{\Delta}}{s = t, P[x/s], \Gamma \Rightarrow \Delta}\ \mathcal{D}_1$	$s = t, P[x/s], \Gamma \Rightarrow \Delta$ $\dashv P[x/t]$ ↖ ∘ $\vdash s = t, P[x/s], \Gamma \Rightarrow P[x/t], \Delta$ (Repl) ↓ $\mathcal{D}_1^* \vdash s = t, P[x/s], P[x/t], \Gamma \Rightarrow \Delta$
L∃	$\dfrac{\overline{A[x/y]}, \overline{\Gamma} \Rightarrow \overline{\Delta}}{\exists x A, \Gamma \Rightarrow \Delta}\ \mathcal{D}_1$	$\exists x A, \Gamma \Rightarrow \Delta \dashv A[x/\mathsf{c}_y]$ ↖ ∘ 2.3.6(ii) ↓ $\mathcal{D}_1^* \vdash A[x/\mathsf{c}_y], \Gamma \Rightarrow \Delta$
R∃	$\dfrac{\overline{\Gamma} \Rightarrow \overline{A[x/t]}, \overline{\exists x A}, \overline{\Delta}}{\Gamma \Rightarrow \exists x A, \Delta}\ \mathcal{D}_1$	$\Gamma \Rightarrow \exists x A, \Delta \dashv A[x/t]$ ↖ $\mathcal{D}_1^* \vdash \Gamma \Rightarrow A[x/t], \exists x A, \Delta$ ↓ ∘ $\vdash A[x/t], \Gamma \Rightarrow \exists x A, \Delta$ (R∃)
L∀	$\dfrac{\overline{A[x/t]}, \overline{\forall x A}, \overline{\Gamma} \Rightarrow \overline{\Delta}}{\forall x A, \Gamma \Rightarrow \Delta}\ \mathcal{D}_1$	$\forall x A, \Gamma \Rightarrow \Delta \dashv A[x/t]$ ↖ ∘ $\vdash \forall x A, \Gamma \Rightarrow A[x/t], \Delta$ (L∀) ↓ $\mathcal{D}_1^* \vdash A[x/t], \forall x A, \Gamma \Rightarrow \Delta$
R∀	$\dfrac{\overline{\Gamma} \Rightarrow \overline{A[x/y]}, \overline{\Delta}}{\Gamma \Rightarrow \forall x A, \Delta}\ \mathcal{D}_1$	$\Gamma \Rightarrow \forall x A, \Delta \dashv \neg A[x/\mathsf{c}_y]$ ↖ ∘ 2.3.6(iii) ↓ $A[x/\mathsf{c}_y]$ ↖ $\mathcal{D}_1^* \vdash \Gamma \Rightarrow A[x/\mathsf{c}_y], \Delta$ ↓ ∘ $\vdash A[x/\mathsf{c}_y], \neg A[x/\mathsf{c}_y], \Gamma \Rightarrow \Delta$ (L¬)

Figure 2.7. Translation of **G3c**$^{=}$ proofs to valuation trees (equational and quantifier rules).

A similar translation is possible from Smullyan's tableaux.

2.4.3 Asymmetry of quantifiers in the valuation tree formal system. Sequent calculus treats the existential quantifier in the antecedent and the general quantifier in the consequent with a symmetrical pair of rules:

$$\mathrm{L}\exists\ \frac{A[x/y], \Gamma \Rightarrow \Delta}{\exists x A, \Gamma \Rightarrow \Delta} \qquad\qquad \mathrm{R}\forall\ \frac{\Gamma \Rightarrow A[x/y], \Delta}{\Gamma \Rightarrow \forall x A, \Delta}$$

for a new variable y representing the witness of $\exists x A$ or the counterexample of $\forall x A$ respectivelly. Tableaux, too, have symmetric rules for $\mathbf{T}\,\exists x A$ and $\mathbf{F}\,\forall x A$.

In our formal system, if we are proving $\exists X A, \Gamma \Rightarrow \Delta$, we can start with a case analysis on $A[X/\mathsf{C}]$ for a new constant C. The right-hand branch is immediatelly closed by Par. 2.3.6(ii). The proof then continues in the left-hand branch, where we assume that $A[X/\mathsf{C}]$ is true (cf. Fig. 2.7 L∃).

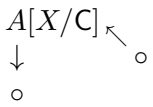
However, if we are proving $\Gamma \Rightarrow \forall X A, \Delta$, we must introduce a new constant C by case analysis on $\neg A[X/\mathsf{C}]$. The right-hand branch is closed by Par. 2.3.6(iii), and the proof continues the left-hand branch where we assume $\neg A[X/\mathsf{C}]$. In order

to get $A[X/\mathsf{C}]$ into the consequent, we need a case analysis on $A[X/\mathsf{C}]$ as seen in Fig. 2.7 R$\forall$.

This two-step process is asymmetric to the treatment of the existential quantifier and somewhat inconvenient. However, since the constant C is interpreted as the counterexample of $\forall X A$, the introduction of C by case analysis on the sentence $\neg A[X/\mathsf{C}]$ is not unnatural. Moreover, using $\neg A[X/\mathsf{C}]$ is necessary. If we changed the condition 2.3.6(iii) to

> (iii*) $p = q +\!\!+ \langle A[X/\mathsf{C}]\rangle +\!\!+ r$ and $\forall X A \in \Delta_q$ for some $\mathsf{C} \in \mathcal{C}$, A, q in which C does not occur, and some r,

and modified the regularity condition Par. 2.3.4(c) accordingly, then the resulting system would be unsound. For instance, the tree



would prove $\exists X A \Rightarrow \forall X A$ with the left-hand child closed by (iii*) and the right-hand child closed by 2.3.6(ii).

2.5 Open-Ended Theory Development in a Valuation Tree

Recall that the development of a verified program in the current version of CL (Par. 2.1.1) consists of definitions of functions and predicates, and of proofs of theorems. Definitions of functions and predicates extend PA, and theorems about functions and predicates are proved within extensions of PA. So, CL has two modes: In the first mode, PA is extended with definitions, and theorems in extensions are postulated. In the second mode, theorems are actually proved. We will now show that with valuation trees and second-order logic, the first mode can be integrated into the second mode, as we have outlined in Par. 2.1.6.

2.5.1 Valuation tree for a series of theorems and theory extensions. Let us fix a language $\mathcal{L}$ and a theory T in $\mathcal{L}$. Assume that $\mathcal{D} \vdash_T \exists X A$. In other words, T proves existence of an object (first-order object, function, or predicate) satisfying A. We can then extend the language $\mathcal{L}$ with a new constant C of the same sort as X, and the theory T with a new axiom $A[X/\mathsf{C}]$. We obtain a new theory T_C in $\mathcal{L}[\mathsf{C}]$. Assume further that we prove some theorems in T_C about C, e.g., $\mathcal{D}_1 \vdash_{T_\mathsf{C}} A_1, \ldots, \mathcal{D}_n \vdash_{T_\mathsf{C}} A_n$.

We can now build the following valuation tree for the empty valuation sequence $\langle\rangle$ in the language $\mathcal{L}$:

$$\begin{array}{l} \mathcal{E} := \exists X\, A \\ \quad\downarrow \\ \quad A[X/\mathsf{C}] \\ \quad\downarrow \\ \quad A_1 \\ \quad\downarrow \\ \quad \vdots \\ \quad A_n \\ \quad\downarrow \\ \quad \circ\ . \end{array} \qquad \begin{array}{l} \nwarrow\ \mathcal{D} \\ \\ \nwarrow\ \circ\ \text{2.3.6(ii)} \\ \\ \nwarrow\ \mathcal{D}_1 \\ \\ \\ \nwarrow\ \mathcal{D}_n \end{array}$$

All leaves in the tree $\mathcal{E}$, except perhaps the left-most one $\langle \exists X A, A[X/\mathsf{C}], A_1, \ldots, A_n \rangle$, are $T, \mathcal{L}$-closed valuation sequences. If the left-most leaf is also closed, then $\mathcal{E}$ proves the empty sequent, which is equivalent to falsity, hence the theory T is inconsistent.

The above development can be iterated further: If we prove in T_{C} existence $\exists X' A'$ of some object satisfying A', and then want to introduce a constant C' for the object, and prove a sequence of theorems about C', we can do that by extending the tree $\mathcal{E}$ at the left-most leaf.

2.5.2 Extensions associated with valuation trees. We would now like to show that the above transition from theory extensions to valuation trees can be reversed. We will prove that if we have a valuation tree for $\langle\rangle$ with exactly one T-open leaf, similar to $\mathcal{E}$ above, than the extension of T with sentences along the path to the open leaf is essentially equivalent to T.

Let us again fix a language $\mathcal{L}$ and a theory T in $\mathcal{L}$. Take a valuation tree $\mathcal{D}$ for the empty valuation sequence in the language $\mathcal{L}$. Assume that all leaves in $\mathcal{D}$ except some p are $T, \mathcal{L}$-closed valuation sequences. Let $\mathcal{C}$ be the finite set of all constants which are not in $\mathcal{L}$ and occur in the single open leaf p of $\mathcal{D}$. Let T, p denote the extension $T, \Gamma_p, \{\, \neg A \mid A \in \Delta_p \,\}$ of T.

The tree $\mathcal{D}$ is a generalization of the tree $\mathcal{E}$ from the previous paragraph, where the only open leaf p was the left-most one, its sequent $\Gamma_p \Rightarrow \Delta_p$ had an empty consequent, and there was only one new constant in $\mathcal{C}$.

The following two theorems and their corollary show that T, p is a very weak extension of T, as weak as extension by definition in first-order logic ([Sho67, §4.6]). First, T, p is a *conservative extension* of T, i.e., for each sentence A of $\mathcal{L}$, we have

$$\vdash_T^{\mathcal{L}} A \text{ iff } \vdash_{T,p}^{\mathcal{L}[\mathcal{C}]} A.$$

Moreover, for each sentence A of $\mathcal{L}[\mathcal{C}]$, there is a sentence A^* of $\mathcal{L}$, called a *translation of A into $\mathcal{L}$*, such that

$$\vdash_{T,p}^{\mathcal{L}[\mathcal{C}]} A \text{ iff } \vdash_T^{\mathcal{L}} A^*.$$

This essentially means that all new constants $\mathcal{C}$ are just convenient abbreviations for their respective existence formulas. Any fact expressible and provable in T, p can be already equivalently expressed and proved in T.

2.5.3 Theorem. *Let $\mathcal{L}$, T, $\mathcal{D}$, p, and $\mathcal{C}$ be as in Par. 2.5.2. Then T, p is a conservative extension of T.*

Proof. Assume $\mathcal{E} \vdash_T^{\mathcal{L}} A$. Without loss of generality, we can assume that no constant from $\mathcal{C}$ occurs in $\mathcal{E}$ (we can rename constants in $\mathcal{E}$ if necessary). Then obviously $\mathcal{E} \vdash_{T,p}^{\mathcal{L}[\mathcal{C}]} A$.

In the opposite direction, assume $\mathcal{E} \vdash_{T,p}^{\mathcal{L}[\mathcal{C}]} A$. Let $\mathcal{E}'$ be a valuation tree obtained by replacing the open leaf p in $\mathcal{D}$ with $\mathcal{E}$, i.e., $\mathcal{E}' = \mathcal{D} \cup \{\, p +\!\!+ q \mid q \in \mathcal{E} \,\}$ (we have used a similar operation in Par. 2.3.13). We would like to show that $\mathcal{E}' \vdash_T A$. We need to consider four different kinds of leaves in $\mathcal{E}'$:

(i) Each leaf $q \in \mathcal{D} \subseteq \mathcal{E}'$, $q \neq p$, is $T, \mathcal{L}$-closed, therefore also $\langle A* \rangle +\!\!+ q$ is $T, \mathcal{L}$-closed.

(ii) If the sequence $\langle A* \rangle +\!\!+ q$ for a leaf $q \in \mathcal{E}$ is $(T, p), \mathcal{L}[\mathcal{C}]$-closed by (Th) for a sentence $B \in \Gamma_p \subseteq (T, p)$, then for the corresponding leaf $p +\!\!+ q \in \mathcal{E}'$, the sequence $\langle A* \rangle +\!\!+ p +\!\!+ q$ is closed by (Ax).

(iii) If the sequence $\langle A* \rangle +\!\!+ q$ for a leaf $q \in \mathcal{E}$ is $(T, p), \mathcal{L}[\mathcal{C}]$-closed by (Th) for a sentence $\neg B \in (T, p)$ where $B \in \Delta_p$, then the sequence $\langle A* \rangle +\!\!+ p +\!\!+ q$ is closed by (R$\neg$).

(iv) If the sequence $\langle A* \rangle +\!\!+ q$ for a leaf $q \in \mathcal{E}$ is $(T, p), \mathcal{L}[\mathcal{C}]$-closed for any other reason than (Th), then $\langle A* \rangle +\!\!+ p +\!\!+ q$ is also $T, \mathcal{L}$-closed for the same reason. □

2.5.4 Theorem. *Let $\mathcal{L}$, T, $\mathcal{D}$, p, and $\mathcal{C}$ be as in Par. 2.5.2. For each sentence A of $\mathcal{L}[\mathcal{C}]$ there is a translation A^* of A into $\mathcal{L}$ such that $\vdash_{T,p} A$ iff $\vdash_{T,p} A^*$.*

Proof. Fix a language $\mathcal{L}$ and a theory T in $\mathcal{L}$. We will work by induction on cardinality of the set of new constants $\mathcal{C}$ occurring in p for all $\mathcal{D}$ and p satisfying the conditions from Par. 2.5.2. For empty $\mathcal{C}$, we simply define $A^* :\equiv A$.

Assume now that $\mathcal{C}$ is non-empty, take any $\mathcal{D}$ with a single open leaf p such that all constants not in $\mathcal{L}$ occurring in p are exactly those in $\mathcal{C}$. By the note in Par. 2.3.4, no member of p introduces two new constants. Therefore, we have $p = p' +\!\!+ \langle B[X/\mathsf{C}] \rangle +\!\!+ p_{\mathsf{C}}$ for a constant $\mathsf{C} \in \mathcal{C}$, a sequence p' with no occurrence of C and occurrences of all constants from $\mathcal{C}' := \mathcal{C} \setminus \{\mathsf{C}\}$, a formula B of $\mathcal{L}[\mathcal{C}']$, and a sequence p_{C}. In other words, $B[X/\mathsf{C}]$ is the first occurrence of C in p, and all constants $\mathcal{C}'$ occur before it. Since p is regular, we have $\exists X B \in \Gamma_{p'}$ or $\forall X B' \in \Delta_{p'}$ for some B' such that $B \equiv \neg B'$. Note that the first occurrence of C in p cannot be a signed formula $B[X/\mathsf{C}]*$ since p would then be closed by 2.3.6(ii) or (iii).

The valuation tree $\mathcal{D}$ can be now divided into trees:

$$\mathcal{D}' := \{\, q \mid p' \not\subset q \in \mathcal{D} \,\}, \qquad \mathcal{D}_{\mathsf{C}} := \{\, q \mid p' +\!\!+ q \in \mathcal{D} \,\}.$$

We have $\mathcal{D} = \mathcal{D}' \cup \{\, p' +\!\!+ q \mid q \in \mathcal{D}_{\mathsf{C}} \,\}$. Note that since p is the only open leaf in $\mathcal{D}$ and p' is a prefix of p, p' is the only open leaf in $\mathcal{D}'$. By the induction hypothesis, for each sentence A of the language $\mathcal{L}[\mathcal{C}']$, there is a translation $A^{*\prime}$ of A into $\mathcal{L}$ such that $\vdash^{\mathcal{L}[\mathcal{C}']}_{T,p'} A$ iff $\vdash^{\mathcal{L}[\mathcal{C}']}_{T,p'} A^{*\prime}$.

Take any sentence A of $\mathcal{L}[\mathcal{C}]$. There are a formula A' of $\mathcal{L}[\mathcal{C}']$ and a variable Y not occurring in B such that $A \equiv A'[Y/\mathsf{C}]$. We define the translation of A into $\mathcal{L}$ as

$$A^* :\equiv \big(\forall Y(B[X/Y] \to A')\big)^{*\prime}.$$

Assume $\vdash^{\mathcal{L}[\mathcal{C}]}_{T,p} A^*$. Since A^* is in $\mathcal{L}$, we have $\vdash^{\mathcal{L}}_{T} A^*$ by the conservativity of T, p, and hence $\vdash_{T,p'} A^*$ by the conservativity of T, p'. By the induction hypothesis, there is a valuation tree $\mathcal{E}$ such that $\mathcal{E} \vdash^{\mathcal{L}[\mathcal{C}']}_{T,p'} \forall Y(B[X/Y] \to A')$. We can prove A in T, p as follows:

$$\begin{array}{l} A \;{}^{\mathcal{L}[\mathcal{C}]}_{T,p}\!\dashv \forall Y(B[X/Y] \to A') \nwarrow \mathcal{E} \\ \quad\downarrow \\ \quad B[X/\mathsf{C}] \to A \nwarrow \circ\,(\mathrm{L}\forall) \\ \quad\downarrow \\ \quad B[X/\mathsf{C}] \nwarrow \circ\,(\mathrm{Th}) \\ \quad\downarrow \\ \quad \circ\,(\mathrm{L}\to)\,. \end{array}$$

For the opposite direction, take a proof $\mathcal{E}$ of A in T, p. We will construct a proof $\mathcal{E}'$ of $\forall Y(B[X/Y] \to A')$ in T, p'. We will then have $\vdash^{\mathcal{L}[\mathcal{C}']}_{T,p'} A^*$ by the induction hypothesis, hence $\vdash^{\mathcal{L}}_{T} A^*$ by the conservativity of T, p', and hence $\vdash^{\mathcal{L}[\mathcal{C}]}_{T,p} A^*$ by the conservativity of T, p. We start building the valuation tree proving $\forall Y(B[X/Y] \to A')$ in T, p' as follows:

$$\begin{array}{l} \mathcal{E}' := \neg(B[X/\mathsf{C}] \to A) \nwarrow \circ\,\text{2.3.6(iii)} \\ \quad\downarrow \\ \quad B[X/\mathsf{C}] \to A \nwarrow A \nwarrow \mathcal{E}'' \\ \quad\downarrow \qquad\qquad\quad\downarrow \\ \quad \circ\,(\mathrm{L}\neg) \qquad\quad \circ\,(\mathrm{R}\to_2)\,. \end{array}$$

We now need to find such a tree $\mathcal{E}''$ that $\mathcal{E}'' \vdash^{\mathcal{L}[\mathcal{C}]}_{(T,p')} \langle B[X/\mathsf{C}] \to A, A* \rangle$. We will derive $\mathcal{E}''$ from another tree

$$\mathcal{E}''' := \mathcal{D}_{\mathsf{C}} \cup \{\, \langle B[X/\mathsf{C}] \rangle +\!\!+ p_{\mathsf{C}} +\!\!+ q \mid q \in \mathcal{E} \,\}.$$

This tree was formed by replacing the leaf $\langle B[X/\mathsf{C}] \rangle +\!\!+ p_{\mathsf{C}}$ in $\mathcal{D}_{\mathsf{C}}$ with $\mathcal{E}$. Let $r := \langle B[X/\mathsf{C}] \to A, A* \rangle +\!\!+ p'$, and observe that $\mathcal{E}''' \vdash^{\mathcal{L}[\mathsf{C}]}_{T,p'} r$:

(i) The root of $\mathcal{D}_{\mathsf{C}}$ is labeled with $B[X/\mathsf{C}]$. Each leaf $\langle B[X/\mathsf{C}]* \rangle +\!\!+ q$ in $\mathcal{D}_{\mathsf{C}}$ is also a leaf in $\mathcal{E}'''$, and the sequence $r +\!\!+ \langle B[X/\mathsf{C}]* \rangle +\!\!+ q$ is closed by $(\mathrm{R}\to_1)$.

(ii) Each leaf $\langle B[X/\mathsf{C}] \rangle +\!\!+ q$ in $\mathcal{D}_{\mathsf{C}}$ for $q \neq p_{\mathsf{C}}$ is also a leaf in $\mathcal{E}'''$. Moreover, $p' +\!\!+ \langle B[X/\mathsf{C}] \rangle +\!\!+ q$ is a leaf in $\mathcal{D}$, and hence it is $T, \mathcal{L}$-closed either by 2.3.6(i), or by 2.3.6(ii) or (iii) for a constant which is not in $\mathcal{L}[\mathcal{C}]$. Therefore, $r +\!\!+ \langle B[X/\mathsf{C}] \rangle +\!\!+ q$ is $(T, p'), \mathcal{L}[\mathcal{C}]$-closed for the same reason.

(iii) For each leaf $q \in \mathcal{E}$, $\langle A* \rangle +\!\!+ q$ is $(T,p),\mathcal{L}[\mathcal{C}]$-closed, and $q' := \langle B[X/\mathsf{C}]\rangle +\!\!+ p_{\mathsf{C}} +\!\!+ q$ is a leaf in $\mathcal{E}'''$. There are three subcases:

1. If $\langle A* \rangle +\!\!+ q$ is closed by (Th) for some $C \in \Gamma_{\langle B[X/\mathsf{C}]\rangle +\!\!+ p_{\mathsf{C}}}$, then $r +\!\!+ q'$ is closed by (Ax).
2. If $\langle A* \rangle +\!\!+ q$ is closed by (Th) for some $\neg C$ where $C \in \Delta_{\langle B[X/\mathsf{C}]\rangle +\!\!+ p_{\mathsf{C}}}$, then $r +\!\!+ q'$ is closed by (Ax).
3. If $\langle A* \rangle +\!\!+ q$ is closed for any other reason, then it is also $(T,p'),\mathcal{L}[\mathcal{C}]$-closed, and so is $r +\!\!+ q'$.

There are no other leaves in $\mathcal{E}'''$.

It remains to build $\mathcal{E}''$ from $\mathcal{E}'''$. We obtain $\mathcal{E}''$ as $\mathcal{E}''_{|p'|}$ from the following inductively defined sequence: $\mathcal{E}''_0 := \mathcal{E}'''$, and for each $i < |p'|$:

$$\mathcal{E}''_{i+1} := \begin{cases} \dfrac{\dfrac{C}{\mathcal{E}_i}\,\circ\,\text{(Th)}}{\circ\,(\text{L}\neg)} & \text{if } p'(|p'|-i-1) = C \text{ for some sentence } C \in \Gamma_{p'}, \\ \dfrac{\dfrac{\dfrac{C}{\mathcal{E}_{i+1}}}{\neg C}\,\circ\,\text{(Th)}}{\circ\,(\text{L}\neg)} & \text{if } p'(|p'|-i-1) = C* \text{ for some sentence } C \in \Delta_{p'}. \end{cases}$$

For each $i \le |p'|$, we have $\mathcal{E}''_i \vdash^{\mathcal{L}[\mathcal{C}]}_{(T,p')} \langle B[X/\mathsf{C}] \to A, A* \rangle +\!\!+ p'_{|p'|-i}$ where p'_k is the initial segment of p of the length k. □

2.5.5 Corollary.

(i) Let $\mathcal{L}$, T, $\mathcal{D}$, p, and $\mathcal{C}$ be as in Par. 2.5.2. For each sentence A of $\mathcal{L}[\mathcal{C}]$ there is a translation A^* of A into $\mathcal{L}$ such that $\vdash_{T,p} A$ iff $\vdash_T A^*$.

(ii) Let T be a theory in a language $\mathcal{L}$, let $\exists X A$ be a sentence of $\mathcal{L}$, and let C be a new constant of the same sort as X. If $\vdash_T \exists X A$, then we have:

- *(a) The theory $T, A[X/\mathsf{C}]$ in $\mathcal{L}[\mathsf{C}]$ is a conservative extension of T.*
- *(b) For each sentence B of $\mathcal{L}[\mathsf{C}]$ there is a translation B^* of B into $\mathcal{L}$ such that $\vdash_{T,A[X/\mathsf{C}]} B$ iff $\vdash_T B^*$.*
- *(c) For each structure $\mathcal{M}$ for $\mathcal{L}$ satisfying T, there is an expansion $\mathcal{M}'$ of $\mathcal{M}$ to $\mathcal{L}[\mathsf{C}]$ satisfying $T, A[X/\mathsf{C}]$.* □

2.6 Existence Axioms in Second-Order Logic

2.6.1 Standard existence axioms. Useful second-order theories contain axioms of existence of second-order objects (functions and predicates) with certain properties beyond the basic ones from initial sequents $(\exists_{f,n})$ and $(\exists_{R,n})$ in Fig. 2.4. As mentioned in the introduction to Sect. 2.2, usually (e.g., [Sim99]), second-order objects are sets. Existence axioms for sets are then universal closures (defined in Par. 2.2.2) of instances of the *comprehension schema*:

$$\exists X \forall x (x \in X \leftrightarrow A)$$

where X does not occur freely in A.

2.6.2 Existence axioms for predicates and functions. In our version of second-order logic, the comprehension schema is straightforwardly adapted to a schema of *existence of predicates with explicit definitions*:

$$\exists R \forall \vec{x}(R(\vec{x}) \leftrightarrow A) \tag{1}$$

for R of all arities and all formulas A in which R does not occur freely. As above, axioms are universal closures of instances of the schema, since some first- and second-order variables may occur freely in A. This schema corresponds to explicit definitions of predicates in first-order logic (see, e.g., [Sho67, §4.6]).

Similarly, contextual definitions of functions in first-order logic can be adapted to a schema of *existence of functions with contextual definitions*:

$$\forall \vec{x}\, \exists y A \wedge \forall \vec{x}\, \forall y \forall z(A \wedge A[y/z] \rightarrow y = z) \rightarrow \exists f \forall \vec{x}\, \forall y(f(\vec{x}) = y \leftrightarrow A) \tag{2}$$

for f of all arities and all formulas A in which f does not occur freely, and z does not occur at all.

For a language $\mathcal{L}$, let $\mathrm{Ex}_{\mathcal{L}}$ denote the set of all existence axioms of the form (1) or (2). It is not difficult to observe, that for any theory T in $\mathcal{L}$ and any structure $\mathcal{M}$ satisfying T, there is a structure $\mathcal{M}'$ for $\mathcal{L}$ satisfying $T, \mathrm{Ex}_{\mathcal{L}}$. Second-order domains $\mathcal{F}_n$, $\mathcal{R}_n$ of $\mathcal{M}$ can be easily iteratively enriched with functions and predicates whose existence is postulated by axioms. The structure $\mathcal{M}'$ is the limit of this iteration.

The theory $T, \mathrm{Ex}_{\mathcal{L}}$ is conservative over the first-order part of the theory T, i.e., any sentence A of $\mathcal{L}$ without second-order quantifiers is a logical consequence of T iff it is a logical consequence of the extension of T with explicit and contextual existence axioms. However, $T, \mathrm{Ex}_{\mathcal{L}}$ is not, in general, a conservative extension of T since it may prove existence of functions of predicates whose existence cannot be proved in T.

2.7 Second-Order Arithmetic for Programming

We intend to use the formal system described in the previous section with a specific second-order extension of Peano arithmetic as the theory T. We define the extension, called CL_2, in Par. 2.7.4. Since our motivation is to use the theory as the foundation of a programming language, it needs to be just strong enough to allow existence of computable functions. Therefore, CL_2 contains only restricted classes of induction axioms and of function and predicate existence axioms.

Just like PA in [Vod01a], CL_2 can be extended with functions for natural encoding of declarative data structures. Existence of clausally defined functions can also be proved. Moreover, second-order theorems are a convenient logical tool to describe modular programming, which the current implementation of CL does not handle satisfactorily (as described in Par. 2.1.2). We outline the modular features enabled by second-order theorems in Sect. 2.8.

Before that, we investigate in Par. 2.7.7 the relationship between CL_2 and the weak subsystem RCA_0 of the standard theory Z_2 of second-order arithmetic.

2.7.1 Motivation for $\mathbf{CL_2}$. The second-order theory CL_2 defined below in Par. 2.7.4 is based on the first-order theory of Peano arithmetic with additional second-order axioms of existence of functions and predicates from a certain restricted class. Basic arithmetic axioms are equivalent to those of Robinson's arithmetic Q, i.e., Peano arithmetic without induction. The particular choice of the language $\mathcal{L}_2$ of CL_2 in Par. 2.7.2 and of the basic axioms in Par. 2.7.4 was influenced by [Sim99].

Induction and second-order axioms of CL_2 are considerably restricted compared to PA and full second-order arithmetic as presented below in Par. 2.7.5. The purpose of this restriction is to allow existence of only computable functions within the theory CL_2. Since primitive recursive functions contain all feasibly computable functions (and much more), the current version of CL was designed so as to allow only definitions of primitive recursive functions. This is achieved by restricting induction in PA to a syntactic class of formulas called Σ_1 (Par. 2.7.3). It has been proved independently by Parsons, Mints, and Takeuti (see, e.g., simple proof-theoretical and model-theoretical proofs in [Fer05, Avi02]) that then exactly primitive recursive functions are provably recursive, i.e., a sentence $\forall x \exists! y A$ is provable by Σ_1 induction iff there is a primitive recursive function f such that for all $n \in \mathbb{N}$, $\mathcal{N}[\mathsf{x} \mapsto n][\mathsf{y} \mapsto f(n)] \vDash A[x/\mathsf{x}][y/\mathsf{y}]$. Here, $\mathcal{N}$ is the standard model of PA whose domain is the set $\mathbb{N}$ of all natural numbers.

In order to enforce this restriction in CL_2, CL_2 will contain induction and function existence axioms for Σ_1 formulas only. Moreover, predicate existence axioms of CL_2 will be restricted to Σ_1 characteristic formulas, i.e., a predicate will exist iff its characteristic function will exist. This corresponds to the implementation of predicates in the current version of CL. Predicate existence axioms for arbitrary Σ_1 formulas would allow existence of recursively enumerable predicates, which is not desirable in a programming language. Moreover, we would obtain an equivalent of a much stronger theory ACA_0 ([Sim99, Rem. I.7.9]).

2.7.2 The language of second-order arithmetic. The language $\mathcal{L}_2$ of second-order arithmetic is a second-order language with first-order constants 0 and 1, binary function constants $+$ and $\cdot$, a binary predicate constant $<$, and first-order, function, and predicate variables as specified in Par. 2.2.1. Note that in accordance with [Sim99], we have included in $\mathcal{L}_2$ the predicate constant $<$ which is usually a defined symbol in PA, and the first-order constant 1 instead of the unary successor function of the language of PA.

Applications of binary constants in terms and formulas of $\mathcal{L}_2$ will be written in infix notation as usual. We will use some common abbreviations such as $s \leq t \equiv s < t \vee s = t$, $2 \equiv 1 + 1$, $t^2 \equiv t \cdot t$.

2.7.3 Classes of formulas by quantifier alternation. Formulas of $\mathcal{L}_2$ are commonly classified by the number of quantifier alternations. We are only interested in the

simplest classes defined below.

For all first-order variables x and terms t in which x does not occur, the following abbreviations are called *bounded quantifiers*

$$\exists x{<}t\, A :\equiv \exists x(x < t \wedge A),$$
$$\forall x{<}t\, A :\equiv \forall x(x < t \to A).$$

We will also use abbreviations $\exists x{\leq}t\, A :\equiv \exists x{<}(t+1)\, A$ and $\forall x{\leq}t\, A :\equiv \forall x{<}(t+1)\, A$.

The smallest class of formulas of $\mathcal{L}_2$ containing atomic formulas, closed under propositional connectives and bounded quantifiers is called Δ_0. The class of formulas of the form $\exists x A$ where A is a Δ_0 formula, is called Σ_1, and the class of formulas of the form $\forall x A$ where A is a Δ_0 formula, is called Π_1.

Note that in formulas from the classes defined above, variables of any sort may occur freely. Such variables are called *parameters*.

2.7.4 Theory $\mathbf{CL_2}$. Taking the motivation outlined in Par. 2.7.1 into account, we will now define the theory CL_2 in the language $\mathcal{L}_2$. Before doing so, let us introduce one more abbreviation, the *unique existential quantifier* for any formula A, as

$$\exists! y A :\equiv \exists y A \wedge \forall y \forall z(A \wedge A[y/z] \to y = z)$$

where z is a first-order variable not occurring in A.

CL_2 is the set of universal closures of:

(i) basic axioms:

$$\begin{aligned} &x + 1 \neq 0, && x + 1 = y + 1 \to x = y,\\ &x + 0 = x, && x + (y + 1) = (x + y) + 1,\\ &x{\cdot}0 = 0, && x{\cdot}(y + 1) = (x{\cdot}y) + x,\\ &\neg\, x < 0, && x < y + 1 \leftrightarrow x < y \vee x = y; \end{aligned}$$

(ii) schema of Σ_1 induction:

$$A[x/0] \wedge \forall x(A \to A[x/x + 1]) \to \forall x A$$

for all Σ_1 formulas A of $\mathcal{L}_2$;

(iii) schema of *existence of functions with contextual* Σ_1 *definitions*:

$$\forall \vec{x}\, \exists! y A \to \exists f \forall \vec{x}\, \forall y(f(\vec{x}) = y \leftrightarrow A)$$

for all Σ_1 formulas A of $\mathcal{L}_2$ in which R does not occur freely;

(iv) schema of *existence of predicates with* Σ_1 *characteristic formulas*:

$$\forall \vec{x}\, \exists! y(A \wedge y < 2) \to \exists R \forall \vec{x}\, (R(\vec{x}) \leftrightarrow A[y/1])$$

for all Σ_1 formulas A of $\mathcal{L}_2$ in which R does not occur freely.

Note that the formulas A in (ii), (iii), and (iv) may contain parameters of any sort.

2.7.5 Second-order arithmetic. We define second-order arithmetic in accordance with [Sim99, §I.2], within our definition of second-order logic. There is a slight technical difference: We use unary predicate variables instead of set variables, and an abbreviation $t \in X \equiv X(t)$ for all first-order terms t.

Second-order arithmetic is a theory Z_2 in $\mathcal{L}_2$ consisting of universal closures of

(i) basic axioms as in 2.7.4(i);

(ii) induction axiom:

$$0 \in R \wedge \forall x(x \in R \to x + 1 \in R) \to \forall x(x \in R);$$

(iii) comprehension schema:

$$\exists R \forall x(x \in R \leftrightarrow A)$$

for all formulas A of $\mathcal{L}_2$ with first-order and unary predicate parameters except R.

A theory T in $\mathcal{L}_2$ is a *subsystem of second-order arithmetic* if axioms of T are theorems of Z_2, i.e., if we have $\mathrm{Z}_2 \vDash T$.

The theory CL_2 contains existence axioms for functions and predicates. Although functions and predicates are definable from sets, we have decided to include them in CL_2 directly, since they are a natural mathematical abstraction of programs. As Z_2 contains only existence axioms for sets, CL_2 is technically not a subsystem of Z_2. The form of existence axioms of CL_2 was chosen simply as an analogy to contextual Σ_1 definitions of functions in first-order logic as described, e.g., [Sho67, §4.6]. We were then surprised to realize that CL_2 is equivalent to an well-known weak subsystem RCA_0 of Z_2. RCA_0 plays an important role in the foundations of mathematics. As we will demonstrate below in Sect. 2.8, CL_2 (and therefore also RCA_0) allows us to describe modular programming. It therefore seems that RCA_0 has an important role also in the theory of programming. We define RCA_0 and show its equivalence with CL_2 in the following paragraphs.

2.7.6 Subsystem $\mathbf{RCA_0}$. The theory RCA_0 ([Sim99, §I.7]) is an extensively studied subsystem of second-order arithmetic. Although RCA_0 is a weak theory, many mathematical concepts and results can be developed in it. They can be characterized as the positive content of recursive analysis and algebra. It can also prove soundness of first-order logic. Details can be found, e.g., in [Sim99, §I.8, Chap. II].

Moreover, RCA_0 is used in reverse mathematics ([Sim99, §I.9]) as a weak basic theory. The goal of reverse mathematics is to find the weakest set of axioms from which a given theorem of ordinary mathematics is provable. In many instances, the set of axioms and the theorem are equivalent and the equivalence is provable in RCA_0. For us, a particularly interesting example of these RCA_0-provable equivalences is the one between completeness of first-order logic and weak König's lemma ([Sim99, §IV.3]).

RCA_0 consists of

(i) basic axioms as in 2.7.4(i);

(ii) schema of Σ_1 induction:

$$A[x/0] \wedge \forall x(A \to A[x/x+1]) \to \forall x A$$

for all Σ_1 formulas A of $\mathcal{L}_2$;

(iii) Δ_1 comprehension schema:

$$(A \leftrightarrow B) \to \exists R \forall x(x \in R \leftrightarrow A)$$

for all Σ_1 formulas A and Π_1 formulas B of $\mathcal{L}_2$ in which R does not occur freely.

Formulas A and B in (ii) and (iii) may contain first-order and unary predicate parameters.

2.7.7 Equivalence of $\mathbf{CL_2}$ and $\mathbf{RCA_0}$. We will show that CL_2 and RCA_0 can interpret each other in the sense of [Sho67, §4.7]. The languages and basic axioms of the two theories are identical. The induction axioms of RCA_0 are a proper subset of induction axioms of CL_2, since RCA_0 admits only first-order and unary predicate parameters in the induction formula. The main difference between the two theories lies in their existence axioms. We will show that each of the two theories proves the other's existence axioms for unary predicates (sets), and that CL_2's existence axioms for functions and for n-ary predicates with $n > 1$ are inessential.

Let us first show that CL_2 proves existence axioms of RCA_0, i.e.,

$$\vdash_{\mathrm{CL}_2} \forall \vec{X}\, \forall x((A \leftrightarrow B) \to \exists R \forall x(x \in R \leftrightarrow A)) \tag{1}$$

for any Σ_1 formula A and any Π_1 formula B of $\mathcal{L}_2$ in which R does not occur freely. There are variables u and v, and Δ_0 formulas A_0 and B_0 such that $A \equiv \exists u A_0$ and $B \equiv \forall v B_0$. Assume without loss of generality that u does not occur in B, and take a first-order variable y not occurring in A nor in B. We construct a Σ_1 characteristic formula A_χ for A:

$$A_\chi :\equiv \exists u(A_0 \wedge y = 1 \vee \neg B_0[v/u] \wedge y = 0).$$

Let $\vec{X}$ be all variables free in A_χ, A and B excluding x and y. We will describe a proof of (1) informally. The proof can be formalized, e.g., in the formal system from Sect. 2.3, in a straightforward fashion. Let $\vec{\mathsf{X}}$ and x be new constants, and let $\bar{F} :\equiv F[\vec{X}/\vec{\mathsf{X}}]$ and $\tilde{F} :\equiv \bar{F}[x/\mathsf{x}]$ for any formula F. Assume $(\tilde{A} \leftrightarrow \tilde{B})$. We have

$$\vdash_{\mathrm{CL}_2} \tilde{A} \leftrightarrow \tilde{B} \Rightarrow \quad (\tilde{A} \leftrightarrow \tilde{A}_\chi[y/1]) \tag{2}$$

$$\vdash_{\mathrm{CL}_2} \tilde{A} \leftrightarrow \tilde{B} \Rightarrow (\neg\tilde{A} \leftrightarrow \tilde{A}_\chi[y/0]). \tag{3}$$

The proof of (3) is the more interesting one: Assume $\neg\tilde{A}$. Then neither $\tilde{A}$ nor $\tilde{B}$ hold, and so $\tilde{B}_0[v/\mathsf{v}]$ does not hold for a new constant v. Consequently, $\neg\tilde{B}_0[v/u][u/\mathsf{v}] \wedge 0 = 0$ holds, hence we have $\tilde{A}_\chi[y/0]$. In the opposite direction, assume $\tilde{A}_\chi[y/0]$. We then have $\neg\tilde{B}_0[v/u][u/\mathsf{u}]$ for a new constant u (since the left-hand disjunct of $\tilde{A}_\chi[y/0]$ cannot hold because $0 \neq 1$). Therefore, $\tilde{B}_0[v/\mathsf{u}]$ does not hold, and so $\tilde{B}$ does not

hold either. Since $\tilde{A}$ and $\tilde{B}$ are equivalent, we obtain $\neg\tilde{A}$. The proof of (2) is similar but simpler, without the detour through B.

With (2) and (3) proved, we can easily show that

$$\vdash_{\mathrm{CL}_2} \tilde{A} \leftrightarrow \tilde{B} \Rightarrow \forall x \exists! y(\bar{A}_\chi \wedge y < 2)$$

by case analysis on the truth value of A_χ. Hence we have $\exists R \forall x(R(x) \leftrightarrow \bar{A}_\chi)$ by Par. 2.7.4(iv), and we obtain $\exists R \forall x(x \in R \leftrightarrow \bar{A})$ by (1).

Interpreting CL_2 in RCA_0 is more involved, since the axioms of RCA_0 contain only first-order and set (i.e., unary predicate) variables. We will

(i) reduce in CL_2 all n-ary predicates for $n > 1$ and all functions of CL_2 to sets;
(ii) show that CL_2 axioms of existence of sets are provable in RCA_0.

(i) In order to reduce predicate and functions to unary predicates, we introduce pairing (cf., [Sim99, §II.2]). Let (x, y) abbreviate the *pairing* term $(x + y)^2 + x$. We have

$$\vdash_T \forall x \forall y(x \leq (x, y) \wedge y \leq (x, y))$$
$$\vdash_T \forall x \forall y \forall u \forall v((x, y) = (u, v) \to x = u \wedge y = v)$$

for $T = \mathrm{CL}_2$ as well as for $T = \mathrm{RCA}_0$. Further, let $(x_1)_1 = x_1$, and for each $n > 0$, let $(x_1, x_2, \ldots, x_{n+1})_{n+1} \equiv (x_1, (x_2, \ldots, x_{n+1})_n)$.

Our next task is to eliminate all n-ary predicate variables for $n > 1$ and all function variables from formulas of $\mathcal{L}_2$. Let us enumerate all n-ary predicate variables of $\mathcal{L}_2$ into a sequence $\{R_{n,i}\}_{i=0}^{\infty}$, and all n-ary function variables of $\mathcal{L}_2$ into $\{f_{n,i}\}_{i=0}^{\infty}$. We can then uniquely map all function and predicate variables to unary ones: $R^1_{n,i} :\equiv R_{1,(2\cdot n-1,i)}$ and $f^1_{n,i} :\equiv f_{1,(2\cdot n,i)}$. We can then define a translation of formulas of $\mathcal{L}_2$ which produces a formula A^1 equivalent to A. In A^1, all second-order variables X are replaced with X^1 in quantifiers, and all applications $X(t_1, \ldots, t_n)$ are replaced with $X^1((t^1_1, \ldots, t^1_n)_n)$.

Another translation A^* eliminates unary function variables by inductively replacing each atomic formula $P[y/f_{1,(2\cdot n,i)}(t)]$ where t does not contain any function variables with $\exists y(R_{1,(2\cdot n,i)}((t, y)) \wedge P^*)$. The translation is extended to all formulas in the obvious way, which also replaces $f_{1,(2\cdot n,i)}$ in quantifiers with $R_{1,(2\cdot n,i)}$.

The combined translation replaces all predicate and function variables with unary predicate variables while maintaining $\vdash_{\mathrm{CL}_2} \forall \vec{X}(A \leftrightarrow A^{1*})$ for all sentences A of $\mathcal{L}_2$ (cf. [Sho67, §4.6]; by $\forall \vec{X} B$ we mean the universal closure of B). When starting with a Σ_1 formula A, the result of the translation A^{1*} is not Σ_1 syntactically. However, it is not difficult to show by structural induction and using pairing, that A^{1*} is Σ_1 *in* CL_2, i.e., there is a Σ_1 formula $A^{1*\prime}$ such that $\vdash_{\mathrm{CL}_2} \forall \vec{X}(A^{1*} \leftrightarrow A^{1*\prime})$ (cf. [HP93, Def. 1.19]).

In particular, an induction axiom 2.7.4(ii) for a Σ_1 formula A with parameters of any sort is provably equivalent to the induction axiom for the Σ_1 formula $A^{1*\prime}$, which has only first-order and unary predicate parameters. An axiom of existence of a n-ary function with Σ_1 definition 2.7.4(iii) for a formula A is provably equivalent

to the universal closure of

$$\forall \vec{x}\, \exists y A^{1*\prime} \to \exists R \forall \vec{x}\, \forall y \big(R\big((\vec{x})_n, y\big) \leftrightarrow A^{1*\prime}\big),$$

which, in turn, is provable from the existence axiom 2.7.4(iv) for the Σ_1 formula

$$\begin{aligned}&\exists w\, \exists v{\leq}w\, \exists x_1{\leq}w \cdots \exists x_n{\leq}w\, \exists y{\leq}w \\ &\quad \big((v, \vec{x}, y)_{n+2} = w \wedge (A_0 \wedge y' = 1 \vee \neg A_0 \wedge y' = 0)\big)\end{aligned}$$

where A_0 is such that $A^{1*\prime} \equiv \exists v A_0$.

(ii) Now that we have shown that all function variables and n-ary predicate variables for $n > 1$ can be eliminated in CL_2, it suffices to show that in RCA_0 we can prove all existence axioms 2.7.4(iv) of CL_2 for unary predicates. So take any Σ_1 formula $A \equiv \exists v A_0$ and assume that the universal closure of

$$\forall x \exists! y (A \wedge y < 2) \tag{4}$$

holds. Then the Π_1 formula

$$B :\equiv \forall w\, \forall v{\leq}w\, \forall y{\leq}w((v, y)_2 = w \wedge A_0 \to y = 1)$$

is provably equivalent to $A[y/1]$: Assume $A[y/1]$, and take any w, v and y such that $(\mathsf{v}, \mathsf{y})_2 = \mathsf{w}$ and $A_0[v/\mathsf{v}][y/\mathsf{y}]$ hold. Hence $\exists v A_0[y/\mathsf{y}] \equiv A[y/\mathsf{y}]$ holds, and by the uniqueness part of (4), we have $\mathsf{y} = 1$. In the opposite direction, assume B. By the existence part of (4), there are v and y such that $A_0[v/\mathsf{v}][y/\mathsf{y}]$ holds. Therefore, we have $\mathsf{y} = 1$ by B, and hence $A[y/1]$. We now obtain $\exists R \forall x(R(x) \leftrightarrow A[y/1])$ from Δ_1 comprehension 2.7.6(iii) of RCA_0.

2.7.8 Models of second-order arithmetic. We would now like answer the question which class of functions is admitted by the existence axioms of CL_2, or, equivalently, which class of functions must be contained in every *model* of CL_2, i.e., every structure for the language $\mathcal{L}_2$ satifying the axioms of CL_2.

In the study of second-order arithmetic, there are some special structures and classes of structures for $\mathcal{L}_2$ ([Sim99, §I.2]). An *ω-structure* $\mathcal{M}$ is such a structure for $\mathcal{L}_2$ where

(i) the first-order domain M is $\mathbb{N}$;

(ii) the interpretations $0^{\mathcal{M}}$, $1^{\mathcal{M}}$, $+^{\mathcal{M}}$, $\cdot^{\mathcal{M}}$, $<^{\mathcal{M}}$ of the constants of $\mathcal{L}_2$ are respectively the numbers 0 and 1, addition, multiplication, and the less-then relation on natural numbers;

(iii) the second-order domains for $n > 0$ are some $\mathcal{R}_n \subseteq \mathcal{P}(\mathbb{N}^n)$, and some $\mathcal{F}_n \subseteq \mathbb{N}^{\mathbb{N}^n}$, so that addition and multiplication are in $\mathcal{F}_2$, and the less-then relation is in $\mathcal{R}_2$.

One of ω-structures is the *intended structure* $\mathcal{N}_2$ for $\mathcal{L}_2$, in which the second-order domains for each $n > 0$ are the sets of all n-ary relations on $\mathbb{N}$ and all n-ary functions from $\mathbb{N}^n$ to $\mathbb{N}$ respectively, i.e., $\mathcal{R}_n = \mathcal{P}(\mathbb{N}^n)$ and $\mathcal{F}_n = \mathbb{N}^{\mathbb{N}^n}$.

In [Sim99, Rem. I.7.5, §VIII.1], the ω-models of RCA_0 are characterized as follows: An ω-structure $\mathcal{M}$ satisfies RCA_0 iff

(i) $\mathcal{R}_1 \neq 0$;
(ii) if $A \in \mathcal{R}_1$ and $B \in \mathcal{R}_1$, then also their recursive join $\{\, 2n \mid n \in A \,\} \cup \{\, 2n+1 \mid n \in B \,\}$ is in $\mathcal{R}_1$;
(iii) if $A \in \mathcal{R}_1$ and B is a Turing-reducible to A, then $B \in \mathcal{R}_1$.
A set B is Turing-reducible to A if the characteristic function of B is computable by a Turing machine using the characteristic function of A as an oracle.

Moreover, RCA_0 has a minimum (unique smallest) ω-model in which $\mathcal{R}_1$ is the set REC of all recursive sets of natural numbers, i.e., all sets whose characteristic functions are computable.

The equivalence of CL_2 and RCA_0 from Par. 2.7.7 allows us to conclude that CL_2 also has a minimum ω-model in which for all $n > 0$, $\mathcal{R}_n$ is the set of all recursive relations on $\mathbb{N}^n$, and $\mathcal{F}_n$ is the set of all recursive functions from $\mathbb{N}^n$ to $\mathbb{N}$.

2.7.9 Relationship of CL_2 and PA. Another corollary of the equivalence of CL_2 and RCA_0 is that the first-order part of CL_2 is exactly the fragment $\mathrm{I}\Sigma_1$ of Peano arithmetic (cf. [Sim99, §I.7.6]), i.e., any sentence A without second-order quantifiers is provable in CL_2 iff it is provable in $\mathrm{I}\Sigma_1$. $\mathrm{I}\Sigma_1$ is a first-order theory consisting of the basic arithmetic axioms 2.7.4(i), and the Σ_1 induction axiom 2.7.4(ii) for each Σ_1 formula A with first-order parameters only. The first-order part of each model of CL_2 is a model of $\mathrm{I}\Sigma_1$, and to each model of $\mathrm{I}\Sigma_1$, we can add second-order domains so that CL_2 is satisfied.

As already mentioned in Par. 2.7.1, $\mathrm{I}\Sigma_1$ can prove the existence property $\forall x \exists! y A$ for defining formulas A of exactly primitive recursive functions ([Avi02, Fer05, HP93, §IV.3(a)]). Therefore, CL_2 can also prove existence for exactly primitive recursive functions.

2.7.10 Bootstrapping a programming language from CL_2. The book [Vod01a] describes how the declarative programming language used in the current version of CL can be developed (bootstrapped) in Peano arithmetic, more specifically in the fragment $\mathrm{I}\Sigma_1$. The bootstrapping involves several steps, most notably: reduction of function definitions by primitive recursion to contextual $\mathrm{I}\Sigma_1$ definitions, definition of a pairing function suitable for encoding of tuples and finite sequences, reduction of course of values recursion with measure to primitive recursion, and development of recursive clausal definitions with measure. For details, we refer the reader to [Vod01a, Chaps. 8, 10].

A suitable computational model also needs to be defined. Although any clausal definition of a function can be reduced to a contextual $\mathrm{I}\Sigma_1$ definition, which is computable, actually computing the function from the contextual definition would be infeasible. Similarly, although all data structures are encoded as numbers through the pairing function, it would be infeasible to actually evaluate the pairing function.

Since CL_2 includes $\mathrm{I}\Sigma_1$, the whole bootstrapping from [Vod01a] can be performed in CL_2. Then, the development of a large library of verified programs from [Kom09] can also be carried out in CL_2.

2.8 Modularity in Second-Order Arithmetic

We would now like to present a more complex example of modularity which can be achieved in second-order logic.

2.8.1 Abstract data types. An abstract data type (ADT) is essentially an algebra. ADT is defined by a set of objects and a collection of basic operations which can be performed on objects from the set. Various kinds of abstract data types are used in programming. A good example are containers — sequences, sets, mappings, priority queues, etc. There are many implementations (concrete data types) for each abstract data type. They typically vary in the time and space complexity of basic operations. When developing a program using an abstract data type, the programmer needs to pick a concrete type which implements the most used operations most efficiently.

2.8.2 Sequences. A ubiquitous abstract data type is finite sequence. A simple implementation of finite sequences are lists. Lists enable constant-time construction of a new sequence from a first element and a trailing sequence, and constant-time access to the first element and the trailing sequence of a sequence. The complexity of most other common operations, such as concatenation and access to an element at a given index, is linear in the length of the sequence or the index.

There are various other concrete types implementing sequences in which some operations can be performed more efficiently. For example, finger trees ([HP06]) allow for (amortized) constant-time access to the first and the last element of a sequence (and the respective remainder sequences), and logarithmic-time concatenation, indexing, and splitting at a given index. There are also implementations with, e.g., amortized constant-time concatenation ([Oka96]). The present author has implemented and verified sequences in the current version of CL in his diploma thesis [Kľu01].

2.8.3 Formalization of sequences in $\mathbf{CL_2}$. The properties of basic list-like operations on sequences are easily formalized in CL_2. Let us define these properties by the following formula:

$$\begin{aligned}\text{Sequence} :\equiv\ & \forall s \left(\mathit{Seq}(s) \leftrightarrow s = \langle\rangle \vee \exists a \exists t \left(s = \langle a \,|\, t\rangle \wedge \mathit{Seq}(t)\right)\right) \\ & \wedge \forall g \forall h \forall p \,\exists f \,\forall x \forall a \forall s \left(\mathit{Seq}(s) \rightarrow \right. \\ & \quad f(\langle\rangle, x) = g(x) \\ & \quad \left. \wedge\, f(\langle a \,|\, s\rangle, x) = h\big(a, s, x, f(s, p(a, s, x))\big)\right).\end{aligned}$$

The formula Sequence has three free variables: the predicate variable *Seq*, the first-order variable $\langle\rangle$, and the function variable $\langle\cdot\,|\,\cdot\rangle$. The predicate *Seq* is intended to hold for all numbers which encode sequences. The first conjunct of Sequence asserts that $\langle\rangle$ and $\langle\cdot\,|\,\cdot\rangle$ are sufficient and necessary to construct all sequences. By the second conjunct of Sequence, functions can be defined by recursion on sequences with substitution (p) in a parameter (x). This type of recursion is called *list recursion*.

Note that we are using the same notation as in Par. 2.3.2, but now we work with codes of sequences rather than finite functions as in Par. 2.3.2.

2.8.4 Interfaces and modules. When we prove the sentence

$$\exists \mathit{Seq}\, \exists \langle\rangle\, \exists \langle \cdot \mid \cdot \rangle \text{ Sequence} \tag{1}$$

in CL_2, we instantiate the existential quantifiers with a concrete implementation of sequences. For instance, we can prove Sequence$[\mathit{Seq}/\mathsf{N}][\langle\rangle/0][\langle \cdot \mid \cdot \rangle/\mathsf{J}']$ where the constants are such that

$$\begin{aligned} &\forall x\, \mathsf{N}(x) \\ &\forall a \forall s (\, \mathsf{J}'(a,s) = (a+s)\cdot(a+s+1) \div 2 + a + 1\,). \end{aligned}$$

These constants are easily definable in CL_2. The pairing function J' is a modified version of Cantor's pairing function J, a polynomial bijection of $\mathbb{N}^2$ onto $\mathbb{N}$. We have $\mathsf{J}'(a,s) = \mathrm{J}(a,s) + 1$, so J' is a bijection of $\mathbb{N}^2$ onto $\mathbb{N} \setminus \{0\}$. The first conjunct of Sequence for N, 0, and J' is proved in $\mathrm{I}\Sigma_1$ in, e.g., [Vod01b]. The second conjunct of Sequence is a consequence of the reducibility of course of values recursion with measure to contextual definitions, mentioned in Par. 2.7.10.

More complex efficiently computable instances of (1) can be proved in $\mathrm{I}\Sigma_1$, such as those in [Kľu01].

From a programming language point of view, an existential theorem (such as (1)) is an *interface* of a module, and a proof of the theorem is the *module* proper. Actual definitions of functions and predicates whose existence is asserted in the theorems, and any auxiliary functions and predicates are hidden inside the proof.

2.8.5 Derived operations. For an abstract data type, one can typically define a collection of generally useful derived operations. The specification of derived operations is uniform across all concrete types implementing the ADT. Derived operations can also be uniformly implemented, although such an implementation may not be the best possible in terms of computational complexity.

In the case of sequences, the number of commonly used derived operations is quite large. During an introductory course on declarative programming, which we teach at Comenius University, students define about 25 such operations in CL (see [Kom09, Chap. 6]). The module Data.List of the state-of-the-art functional programming language Haskell ([PJ03, HHL10]) contains around 100 operations, all of which can be defined for any sequence implementation. The module Data.Sequence, which implements sequences as finger trees, reimplements around 70 of the list operations. Around 30 of these operations on finger trees are indeed obtained uniformly through the class (Haskell term for an ADT) Foldable, which, essentially, has list recursion as its basic operation.

2.8.6 Some derived operations on sequences. A formula Seq_ops specifying a few assorted derived operations on sequences is depicted in Fig. 2.8. Seq_ops has the

$$
\begin{aligned}
\text{Seq\_ops} :\equiv\ & \forall a \forall b \forall i \forall s \forall t \big(\mathit{Seq}(s) \wedge \mathit{Seq}(t) \wedge \mathit{Seq}(u) \to \\
& \quad \langle\rangle +\!\!+ t = t \\
& \wedge \langle a \,|\, s \rangle +\!\!+ t = \langle a \,|\, s +\!\!+ t \rangle \\
& \wedge L\langle\rangle = 0 \\
& \wedge L\langle a \,|\, s \rangle = L(s) + 1 \\
& \wedge \mathit{Rev}\langle\rangle = \langle\rangle \\
& \wedge \mathit{Rev}\langle a \,|\, s \rangle = \mathit{Rev}(s) +\!\!+ \langle a \,|\, \langle\rangle \rangle \\
& \wedge (s +\!\!+ \langle a \,|\, t \rangle)_{L(s)} = a \\
& \wedge (s +\!\!+ \langle a \,|\, t \rangle)[L(s) := b] = s +\!\!+ \langle b \,|\, t \rangle \\
& \wedge (i \geq L(s) \to s[i := b] = s) \\
& \wedge \mathit{Take}(L(s), s +\!\!+ t) = s \\
& \wedge (i > L(s) \to \mathit{Take}(i, s) = s) \\
& \wedge \mathit{Drop}(L(s), s +\!\!+ t) = t \\
& \wedge (i > L(s) \to \mathit{Drop}(i, s) = \langle\rangle) \\
& \wedge \mathit{Split}(L(s), s +\!\!+ t) = (s, t) \\
& \wedge (i > L(s) \to \mathit{Split}(i, s) = (s, \langle\rangle)) \\
& \wedge \forall f \exists \mathit{Map} \forall p \forall a \forall s \big(\mathit{Seq}(s) \to \\
& \qquad \mathit{Map}(p, \langle\rangle) = \langle\rangle \\
& \quad \wedge \mathit{Map}(p, \langle a \,|\, s \rangle) = \langle f(p, a) \,|\, \mathit{Map}(p, s) \rangle \big) \big)
\end{aligned}
$$

Figure 2.8. Example of a parameterized module – derived operations on sequences.

free variables of Sequence for the basic operations, and free variables for the derived operations: concatenation $+\!\!+$, the length of a sequence L, sequence reversal *Rev*, indexing $(\cdot)_\cdot$, replacement of the element at a given index $\cdot[\cdot := \cdot]$, a function *Take* yielding the initial segment of a sequence up to a given index, a dual function *Drop* yielding the trailing part of a sequence from a given index, and a function *Split* which yields a pair whose components are the values of the previous two operations.

2.8.7 Parameterized modules. Within a proof of the sentence

$$
\begin{aligned}
& \forall \mathit{Seq} \, \forall \langle\rangle \, \forall \langle \cdot \,|\, \cdot \rangle \; \exists +\!\!+ \, \exists L \, \exists \mathit{Rev} \, \exists (\cdot)_\cdot \, \exists \cdot[\cdot := \cdot] \, \exists \mathit{Take} \, \exists \mathit{Drop} \, \exists \mathit{Split} \\
& \qquad (\text{Sequence} \to \text{Seq\_ops})
\end{aligned}
\tag{1}
$$

existential quantifiers will be instantiated with implementations of derived operations parameterized by the sequence predicate *Seq*, the basic sequence operations $\langle\rangle$ and $\langle \cdot \,|\, \cdot \rangle$, and the schema of list induction.

In the programing terminology, a $\forall\exists$-theorem is an *interface of a parameterized module*, and the proof of this theorem is the *parameterized module* proper.

Note that a parameterized module which proves (1) contains a nested parameterized module. Namely, existence of a function *Map* is asserted for any function f to be applied to the elements of a sequence.

2.8.8 Arithmetization of completeness of second-order logic. In CL_2, one can prove equivalence of an arithmetized version of the completeness theorem for second-order logic with the weak König's lemma (cf. [Sim99, Thm. I.10.3]). We have written a short case study [Kľu10] outlining a modular version of the proof in CL_2 and the valuation tree calculus. We would like to eventually prove completeness formally in the new implementation of CL.

3 Congruence Closure

We have established in the previous chapter a general logical framework for the new implementation of CL. The framework consists of the valuation tree formal system and the second-order theory CL_2. The manual part of the development of verified programs in the new implementation of CL will be performed within this framework. By the manual part of development, we mean definition of modules, their constituting functions, statement of their properties, and construction of proofs of these properties. Since we have integrated theory extensions into proofs in Sect. 2.5, the whole development will, in fact, be carried out in the proof-building part of the new CL proof assistant.

We will now turn our attention to the automatic part of the proof assistant. The automatic part can be called by the proof-building part from any node of a valuation tree with a sequent $\Gamma \Rightarrow \Delta$. The automatic part tries to prove this sequent. We are interested in supporting the human user by proving simple sequents quickly. We do not want to slow down the user's progress by excessively long attempts at automatic proofs, nor confuse the user with unexpected rewriting. It is not our ambition to have an automatic part which could prove any theoretically provable sequent.

Automation of equality reasoning is a natural requirement on a proof assistant. Equality plays a central role in mathematical reasoning in general, and in verification of programs in particular. Manual application of equality rules (or, in the case of our formal system from Sect. 2.3, a combination of case analysis and closed sequents (Refl) and (Repl)) would be very tedious.

We will therefore start, in this chapter, with the congruence closure algorithm, which automates basic equality reasoning. This algorithm can decide validity of sequents $\Gamma \Rightarrow \Delta$ which contain only closed equalities. In the following two chapters, we will explore the possibilities of extending the congruence closure algorithm in order to decide sequents containing more complex formulas. We are especially interested in universally quantified Horn clauses since CL programs are conjunctions of such clauses. Moreover, experience with verification of programs ([Kom09]) shows, that also many properties of programs can be expressed as universally quantified Horn clauses.

The aim of this chapter is to describe and prove the congruence closure algorithm in detail. We will characterize the algorithm in logical and set-theoretical terms, rather than using the graph-theoretical or term-rewriting terminology found in literature ([NO80, DST80, NO03]). The presentation of the algorithm on set-theoretical trees in Sect. 3.5 is, in our opinion, more intuitive than on graphs. Moreover, the characterization through a proof system from Sect. 3.4 extends well to more general algorithms in chapters 4 and 5.

3.1 Overview of the Congruence Closure Algorithm

3.1.1 Purpose and operation of the congruence closure algorithm. The congruence closure algorithm was invented in late 1970s. Several versions and proofs of the algorithm exist, e.g., [NO80, DST80, NO03]. The algorithm can derive equational logical consequences from a set of closed (i.e., variable-free) equalities in time $O(n \log n)$.

The algorithm is given a finite set of closed equalities $\Gamma_0 = \{s_1 = t_1, \ldots, s_n = t_n\}$, a closed equality $s_0 = t_0$, and decides whether $s_0 = t_0$ is a logical consequence of Γ_0. Let us denote by D the set of all subterms of $\vec{s}$, $\vec{t}$, s_0, t_0. Two data structures are used in the algorithm: (i) an oriented graph representing all terms in D; (ii) a data structure (usually one supporting the union–find-set algorithm of [Tar75]) representing a relation of equivalence on D initialized so that each term is in its own equivalence class. The algorithm iterates through the equalities in the set Γ_0. For each i, $1 \leq i \leq n$, the equivalence classes of s_i and t_i are merged, and the term-representing graph is then used to close the equivalence-representing data structure under *congruence*, i.e., the property

$$u_1 = v_1 \wedge \cdots \wedge u_k = v_k \rightarrow f(\vec{u}) = f(\vec{v})$$

for all terms $f(\vec{u})$, $f(\vec{v})$ in D. Typically, new equalities inferred from congruence are enqueued, and processed after all equalities from the initial set Γ_0.

The algorithm terminates when all equalities from Γ_0 have been considered and no new equalities can be derived from congruence. Then the equivalence-representing data structure is queried to determine whether s_0 and t_0 are in the same equivalence class.

There is also an "online" version of the algorithm, in which data structures are constructed during the iteration through Γ_0.

3.1.2 Congruence closure in the current version of CL. The automatic part of the proof assistant in the current version of CL is based on Shostak's method of combination of decision procedures (see, e.g., [RS01]). As already mentioned in the introduction, the main principle of this method is rewriting of terms to a canonical form. This principle is also applied to congruence closure. Equalities in Γ_0 are oriented and applied to s_0 and t_0 as rewriting rules. If the two terms are rewritten to the same canonical form, they are equal. The products of rewriting can be more complex and less intuitive than the original terms.

3.1.3 Our approach to congruence closure. Existing formulations and proofs of the congruence closure algorithm are based either on graph ([NO80, DST80]) or term-rewriting terminology ([NO03]). We will describe the algorithm in logical terms, and in a way which will be applicable to extensions presented in later chapters.

Our version of the algorithm will use a different representation of terms than the classical versions of [NO80, DST80], which use directed graphs. In [NO03], terms are simplified by currying, i.e., each $f(t_1, t_2, \ldots, t_k)$ is replaced by $(((f\ t_1)\ t_2) \ldots\ t_k)$, and flattened by introducing new equalities with new constants $(f\ t_1) = \mathsf{c}_1$, $(\mathsf{c}_1\ t_2) = \mathsf{c}_2$,

$\ldots$, $(\mathsf{c}_{k-1}\ t_k) = \mathsf{c}_k$. While this approach simplifies the algorithm, it also obscures the structure of terms.

We will represent terms by sequences $\langle f, [t_1], [t_2], \ldots, [t_n], \emptyset\rangle$, where $[t]$ is the equivalence class of a term t. Such sequences allow us to define a set-theoretical tree ordered by the suffix relation. Congruence closure can be intuitively viewed as merging of branches of this tree.

The congruence closure algorithm considers equalities from the input set and new equalities implied by congruence sequentially. We will make syntactic distinction between already considered and still pending equalities by wrapping the latter in atomic clauses defined in Par. 3.2.1. The notion of clauses will also be useful later.

The already considered equalities induce an equivalence relation on closed terms which does not necessarily satisfy all instances of congruence. We will define a semantics of a quantifier-free fragment of second-order logic under such relations in Sect. 3.3. This semantics is somewhat similar to Herbrand interpretations used in the theory of logic programming (see, e.g., [Apt88]). The relations-based semantics will help us characterize the intermediate states of the congruence closure algorithm, and we will also use it in the following chapter.

The congruence closure works with equalities only, but can be easily extended to all atomic sentences if we replace predicate symbols with new function symbols. The necessary equivalent transformation is described in Par. 3.2.2 and proved in Thm. 3.2.3.

After this preliminary work, we will introduce a two-rule formal system for logical characterization of the congruence closure algorithm in Sect. 3.4. Sect. 3.5 will present an abstract, set-theoretical version of the algorithm. In Sect. 3.6, we will discuss an efficient imperative implementation. We will conclude this chapter with notes on related work.

3.2 Quantifier-Free Sentences and Functional Languages

3.2.1 Closed terms, quantifier-free sentences, clauses. We use the notion of a second-order language as defined in Par. 2.2.1. We will consider first-order constants c to be function constants of arity 0 (i.e., *nullary*). We will occasionally need to make a distinction between a first-order constant c itself, and the term (a sequence of symbols with one element) constructed from that constant. We will then write the term as $c()$. A *function symbol* is a function constant of any arity $n \geq 0$. A *predicate symbol* is a predicate constant of any arity. Function symbols will be denoted by the letters f, g, h, predicate symbols by P, Q, R. We will not deal with function nor predicate variables, and the word *variable* will mean a first-order variable.

We will assume that every language $\mathcal{L}$ contains at least one nullary function symbol, which we will denote by o. This assumption is needed so we can form closed terms. The set of all closed terms of $\mathcal{L}$ will be denoted by $D_{\mathcal{L}}$. It is the smallest set consisting of expressions $f(s_1, \ldots, s_n)$, where f is a function symbol of $\mathcal{L}$ of arity $n \geq 0$, and s_1, $\ldots$, s_n are closed terms.

Atomic sentences of the language $\mathcal{L}$ are either *closed equalities* $s = t$ with $s, t \in D_{\mathcal{L}}$, or applications of predicate symbols $P(\vec{s})$ of arity $n \geq 1$ with $\vec{s} \in D_{\mathcal{L}}$. *Quantifier-free sentences* of $\mathcal{L}$ are formed from atomic sentences by the connectives $\neg$, $\wedge$, $\vee$, and $\rightarrow$. We will include among quantifier-free sentences also sequents $\Gamma \Rightarrow \Delta$ (as defined in Par. 2.3.3) for finite sets Γ, Δ of quantifier-free sentences. Since Γ and Δ are sets, sequents are more flexible than ordinary propositional connectives. The satisfaction of sequents in structures was defined in Par. 2.3.3, but note that now Γ and Δ can also contain nested sequents.

If Γ and Δ contain only atomic sentences, the sequent $\Gamma \Rightarrow \Delta$ is called a *closed clause*. As a special case, sequents $\Rightarrow s = t$ are called *atomic clauses* (some authors also define *unit clauses* of two kinds: *positive* $\Rightarrow s = t$, and *negative* $s = t \Rightarrow$).

In automated deduction and logic programming, the adjective "*ground*" is used by authors instead of "closed and quantifier-free".

Note that until Chapter 5, we will deal **exclusively** with closed terms, closed equalities, closed clauses, and quantifier-free sentences. We will usually call them just terms, equalities, clauses, and sentences respectively. We use the usual notation for terms and formulas (see Par. 2.2.2).

3.2.2 Functional languages. A language $\mathcal{L}$ is *functional* if it does not contain any predicate symbols. The word "functional" is meant as an adjective derived from "function," and does not refer to higher-order functions. Note that equality is a logical symbol, not a predicate. Since the congruence closure is centered around equality, it is natural that from now on, we will deal **exclusively** with functional languages. We do not loose the expressivity of general first-order languages by such a restriction. We justify this claim by the following definitions and by Thm. 3.2.3.

For a first-order language $\mathcal{L}$, we designate by $\mathcal{L}_*$ the functional language containing all function symbols of $\mathcal{L}$, and for every n-ary predicate symbol P of $\mathcal{L}$ a new n-ary function symbol P_*. For each formula A of $\mathcal{L}$, the *translation* A_* *of* A *to* $\mathcal{L}_*$ is obtained by replacing every occurrence of an atomic formula $P(\vec{s})$ in A with the equality $P_*(\vec{s}) = \mathsf{o}$. The function symbols P_* play thus the role of the characteristic functions of predicates P.

3.2.3 Theorem (Elimination of predicate symbols). *For a language $\mathcal{L}$, its functional language $\mathcal{L}_*$, and a quantifier-free sentence A of $\mathcal{L}$, we have $\vDash A$ iff $\vDash A_*$.*

Proof. In the direction ($\leftarrow$), we assume $\vDash A_*$, and take any structure $\mathcal{M}$ for $\mathcal{L}$. If the first-order domain M of $\mathcal{M}$ has at least two elements, we designate by c an element of M different from $\mathsf{o}^{\mathcal{M}}$. We define the structure $\mathcal{M}_*$ for $\mathcal{L}_*$ with the same domain and the same interpretations of function symbols. For a predicate symbol P, we interpret

$$P_*^{\mathcal{M}_*}(\vec{x}) = \begin{cases} \mathsf{o}^{\mathcal{M}} & \text{if } (\vec{x}) \in P^{\mathcal{M}}, \\ c & \text{otherwise.} \end{cases} \tag{1}$$

We then prove $\mathcal{M}_* \models B_*$ iff $\mathcal{M} \models B$ by a straightforward induction on the structure of sentences B in $\mathcal{L}$. From $\mathcal{M}_* \models A_*$, we then obtain $\mathcal{M} \models A$.

If the domain of $\mathcal{M}$ consists of a single element $\mathsf{o}^{\mathcal{M}}$, we construct $\mathcal{M}_*$ for $\mathcal{L}_*$ with a two element domain $\{c, \mathsf{o}^{\mathcal{M}}\}$ for some $c \neq \mathsf{o}^{\mathcal{M}}$. We interpret the function symbols f (including the symbols P_*) in $\mathcal{M}_*$ to yield c if at least one argument is c. We interpret $P_*^{\mathcal{M}_*}(\mathsf{o}^{\mathcal{M}}, \ldots, \mathsf{o}^{\mathcal{M}})$ as in (1). For any other f, we let $f^{\mathcal{M}_*}(\mathsf{o}^{\mathcal{M}}, \ldots, \mathsf{o}^{\mathcal{M}}) = \mathsf{o}^{\mathcal{M}}$. The rest of the proof is similar as above.

For the proof in the direction ($\rightarrow$), we assume $\models A$, and take any structure $\mathcal{M}_*$ for $\mathcal{L}_*$. We define the structure $\mathcal{M}$ for $\mathcal{L}$ with the same domain and with the same interpretation of function symbols of $\mathcal{L}$. For a predicate symbol P of $\mathcal{L}$, we define $P^{\mathcal{M}}(\vec{x})$ to hold iff $\mathcal{M}_* \models P_*(\vec{x}) = \mathsf{o}$. We then continue similarly as above. □

3.3 Binary Relations and Satisfaction of Sentences

3.3.1 Binary relations over terms. We will be investigating propositional consequence for functional languages $\mathcal{L}$ under various properties of the equality symbol. Since satisfaction of sentences in structures presupposes the standard properties of equality, we need to define the semantics of equality by weaker means.

The role of structures will be played by binary relations $\simeq$ over $D_{\mathcal{L}}$. A relation $\simeq$ is an *equivalence* if it is

(i) reflexive ($s \simeq s$ for all $s \in D_{\mathcal{L}}$),
(ii) symmetric (for all $s, t \in D_{\mathcal{L}}$, if $s \simeq t$, then $t \simeq s$), and
(iii) transitive (for all $s, t, u \in D_{\mathcal{L}}$, if $s \simeq t$ and $t \simeq u$, then $s \simeq u$).

The relation $\simeq$ is a *congruence* if it is an equivalence respecting the function symbols of $\mathcal{L}$, i.e., if

(iv) for every function symbol f of $\mathcal{L}$ of arity $n \geq 1$, and all terms of $D_{\mathcal{L}}$ such that $s_1 \simeq t_1$, …, $s_n \simeq t_n$, we have $f(s_1, \ldots, s_n) \simeq f(t_1, \ldots, t_n)$.

3.3.2 Satisfaction relation for functional languages. Let $\mathcal{L}$ be a functional language and $\simeq$ a relation over $D_{\mathcal{L}}$. We define the relation $\simeq$ *satisfies* A, in writing $\simeq \models A$, for sentences A of $\mathcal{L}$ as follows:

(i) $\simeq \models s = t$ iff $s \simeq t$,
(ii) $\simeq \models \neg A$ iff $\simeq \not\models A$,
(iii) $\simeq \models A \wedge B$ iff $\simeq \models A$ and $\simeq \models B$,
(iv) $\simeq \models A \vee B$ iff $\simeq \models A$ or $\simeq \models B$,
(v) $\simeq \models A \rightarrow B$ iff $\simeq \models B$ whenever $\simeq \models A$,
(vi) $\simeq \models \Gamma \Rightarrow \Delta$ iff there is a $B \in \Delta$ such that $\simeq \models B$ whenever $\simeq \models A$ for all $A \in \Gamma$.

For a set of sentences T, we say that $\simeq$ *satisfies* T, in writing $\simeq \models T$, if $\simeq$ satisfies each $A \in T$. For $q = \mathrm{p}, \mathrm{e}, \mathrm{c}$ and $\simeq$ satisfying T, we say that $\simeq$ is a *q-model* of T if $\simeq$ is in the order p, e, c, any relation, an equivalence, or a congruence. We say that A is a *q-consequence of* T, in writing $T \models_q A$, iff every q-model $\simeq$ of T satisfies A.

Note that if $S \subseteq T$ and $S \models_q A$, then $T \models_q A$, since every q-model of T is necessary a q-model of S. For a set S of sentences we write $T \models_q S$ when $T \models_q A$ for all $A \in S$.

3.3.3 Conservative extensions. For $q = \mathrm{p}, \mathrm{e}, \mathrm{c}$, we call a theory T_1 in $\mathcal{L}_1$ a *q-extension* of the theory T_2 in $\mathcal{L}_2$ if $\mathcal{L}_1$ is an extension of $\mathcal{L}_2$, and $T_1 \models_q T_2$ holds. T_1 *is q-conservative over T_2 for S* if T_1 is a q-extension of T_2, and whenever $T_1 \models_q A$ for $A \in S$, then already $T_2 \models_q A$.

3.3.4 Restrictions and extensions of relations over terms. For a functional language $\mathcal{L}$, let D be a possibly infinite subset of $D_{\mathcal{L}}$. D is *downward closed* if whenever $f(s_1, \ldots, s_n) \in D$, then also $s_1, \ldots, s_n \in D$. We call such a D a *syntactic domain*, or just *domain* if no confusion with domains of structures can arise. We call a relation $\simeq$ over D e-*closed* if it is an equivalence over D. An equivalence over D is c-*closed* if for every $f(\vec{s}), f(\vec{t}) \in D$ such that $\vec{s} \simeq \vec{t}$, we have $f(\vec{s}) \simeq f(\vec{t})$.

For any set T of sentences of $\mathcal{L}$, the set of all subterms occurring in the sentences of T is clearly a domain, and we call it the *domain of T*. Coversely, if D is a domain, a formula A is called *D-bounded* if all closed terms occurring in A are from D. A set of formulas is D-bounded if all its members are.

For an equivalence $\simeq$ over a domain D, we refer to the set $\{\, s \mid s \simeq t \,\}$ as the *equivalence class* of the term $t \in D$, and denote it by $[t]_{\simeq}$. We drop the subscript when the equivalence is known from the context. It is well known that for every equivalence $\simeq$ over a domain D, $\bigcup_{t \in D} [t]_{\simeq} = D$, and for all terms $s, t \in D$, $[s]_{\simeq} \cap [t]_{\simeq} \neq \emptyset$ iff $[s]_{\simeq} = [t]_{\simeq}$. The set $D/{\simeq} = \{\, [t]_{\simeq} \mid t \in D \,\}$ is the *quotient set* of the set D and the equivalence $\simeq$.

For $q = \mathrm{e}, \mathrm{c}$, the restriction of a q-relation $\simeq$ over $D_{\mathcal{L}}$ to a domain D is clearly a q-closed relation over D. Moreover, it can be easily seen that if two relations $\simeq$ and $\simeq'$ *coincide* on D, i.e., if their restrictions to D are identical, then for every $s, t \in D$, we have $\simeq \models s = t$ iff $\simeq' \models s = t$. This extends to $\simeq \models A$ iff $\simeq' \models A$ for any sentence A with all subterms occurring in D. The following important lemma asserts a property in the opposite direction:

3.3.5 Lemma (Expansion). *Every* c*-closed relation $\simeq$ over a domain D in a functional language $\mathcal{L}$ can be* expanded *to a congruence $\simeq'$ over $D_{\mathcal{L}}$ such that $\simeq$ and $\simeq'$ coincide on D, and for all function symbols g that do not occur in the terms of D, whenever $s \in D$, and g occurs in a term t, then $\simeq' \not\models s = t$.*

Proof. Let $ord\colon D/{\simeq} \to \mathbb{N}$ be an injection into odd natural numbers. We construct the structure $\mathcal{M}$ for $\mathcal{L}$ with the domain a subset of natural numbers consisting of all even numbers and of those odd numbers which are in the range of ord. We interpret an n-ary ($n \geq 0$) function symbol f of $\mathcal{L}$ as follows:

$$f^{\mathcal{M}}(\vec{x}) = \begin{cases} ord([f(\vec{s})]) & \text{if } f(\vec{s}) \in D \text{ and } x_1 = ord([s_1]), \ldots, x_n = ord([s_n]), \\ 2{\cdot}x_1 & \text{otherwise, where if } n = 0, \text{ then } x_1 = 0. \end{cases}$$

This is a legal definition because *ord* is an injection, and $\simeq$ is c-closed. Thus if $f(\vec{s}), f(\vec{t}) \in D$ with $ord([s_i]) = ord([t_i])$ for all $i = 1, \ldots, n$, then also $[s_i] = [t_i]$, and hence $[f(\vec{s})] = [f(\vec{t})]$.

By a simple induction on terms s, we get if $s \in D$, then $s^{\mathcal{M}} = ord([s])$. Consequently for $s, t \in D$, $s \simeq t$ iff $\mathcal{M} \vDash s = t$. For any terms s and t, we define $s \simeq' t$ to hold iff $\mathcal{M} \vDash s = t$. Clearly, $\simeq'$ is a congruence and its restriction to D is $\simeq$. Moreover, for a function symbol g which does not occur in D, by induction on terms t we have if g occurs in t, then $t^{\mathcal{M}}$ is even, hence $s \not\simeq' t$ for all $s \in D$.

Indeed, assume $t = f(\vec{t})$ for some function symbol f and some subterms $\vec{t}$. If $f = g$, then since g does not occur in D, there can be no term $g(\vec{s}) \in D$, so $t^{\mathcal{M}}$ is even: $t^{\mathcal{M}} = 2{\cdot}t_1^{\mathcal{M}}$ or $t^{\mathcal{M}} = 2{\cdot}0$ depending on the arity of g. If $f \neq g$, the arity of f is positive, and g occurs in t_i for some i. By the inductive hypothesis, $t_i^{\mathcal{M}}$ is even, but then there is no $f(\vec{s}) \in D$ such that $t_i^{\mathcal{M}} = ord([s_i])$, hence $t^{\mathcal{M}} = 2{\cdot}t_1$ is even. □

3.3.6 Axioms of equality. Fix a functional language $\mathcal{L}$. For sequences of terms $\vec{s} = s_1, \ldots, s_n$ and $\vec{t} = t_1, \ldots, t_n$, we abbreviate by $\vec{s} = \vec{t}$ the (possibly empty) set $s_1 = t_1, \ldots, s_n = t_n$.

It should be clear that any equivalence over $D_{\mathcal{L}}$ is an e-model of the set $\mathrm{E}^{\mathcal{L}}$ of all sentences

$$s = s \qquad s = t \to t = s \qquad s = t \wedge t = u \to s = u \tag{1}$$

and vice versa, any relation $\simeq$ satisfying $\mathrm{E}^{\mathcal{L}}$ is an equivalence.

If $\simeq$ is a congruence, then it is also a c-model of the set $\mathrm{C}^{\mathcal{L}}$ of all clauses

$$\vec{s} = \vec{t} \Rightarrow f(\vec{s}) = f(\vec{t}) \tag{2}$$

where $f \in \mathcal{L}$ is of arity $n > 0$. Vice versa, any equivalence satisfying $\mathrm{C}^{\mathcal{L}}$ is a congruence.

When $T \vDash_{\mathrm{c}} A$ holds, we say that A *is a quasi-tautological consequence of* T (A *is a quasi-tautology* if T is empty). The above discussion of axioms of equality is formally expressed in Thm. 3.3.7. Theorem 3.3.8(i) connects the standard tautological consequence of logic with our relation $\vDash_{\mathrm{p}}$. It follows that the standard logical definition of quasi-tautological consequence as a *tautological consequence from the axioms of equality* (cf. [Sho67]) agrees with our definition above. For the sake of completeness, we prove in 3.3.8(ii) the standard connection between quasi-tautological and logical consequence for quantifier-free sentences.

3.3.7 Theorem (Reductions of *q*-consequence). *For a quantifier-free theory T in a functional language $\mathcal{L}$ and a quantifier-free sentence A, we have:*

(i) $T \vDash_{\mathrm{e}} A$ *iff* $T, \mathrm{E}^{\mathcal{L}} \vDash_{\mathrm{p}} A$,

(ii) $T \vDash_{\mathrm{c}} A$ *iff* $T, \mathrm{C}^{\mathcal{L}} \vDash_{\mathrm{e}} A$. □

3.3.8 Theorem (Tautological and quasi-tautological consequence). *For a quantifier-free theory T in a functional language $\mathcal{L}$ and a quantifier-free sentence A, we have:*

(i) A is a tautological consequence of T iff $T \models_{\mathrm{p}} A$,
(ii) $T \models A$ iff $T \models_{\mathrm{c}} A$.

Proof. (i) In the direction ($\rightarrow$), assume that A tautologically follows from T, and take any p-model $\simeq$ of T. Construct an assignment I of truth values to equalities $s = t$ of $\mathcal{L}$ as $I(s = t) = \mathfrak{t}$ if $s \simeq t$, and $I(s = t) = \mathfrak{f}$ otherwise. We then extend this to any B of $\mathcal{L}$ by showing that B is satisfied in I iff $\simeq \models B$. The assignment I satisfies T, and so A is true in I, and hence $\simeq \models A$. In the direction ($\leftarrow$), we assume $T \models_{\mathrm{p}} A$, and take any assignment I satisfying T. For $s = t$ in $\mathcal{L}$, we define $s \simeq t$ to hold iff $I(s = t) = \mathfrak{t}$, and continue similarly as above.

(ii) The proof is structurally similar to that of (i). In the direction ($\rightarrow$), we take any c-model $\simeq$ of T, and construct a structure $\mathcal{M}$ for $\mathcal{L}$ with the domain $D_{\mathcal{L}}/{\simeq}$, where for a function symbol f of $\mathcal{L}$, we define $f^{\mathcal{M}}([s_1], \ldots, [s_n]) = [f(s_1, \ldots, s_n)]$. We can see, similarly as in the proof of Lemma 3.3.5, that this is a legal definition. It is easy to see that $s^{\mathcal{M}} = [s]$, and so $\mathcal{M} \models s = t$ iff $s \simeq t$. In the direction ($\leftarrow$), we take any model $\mathcal{M}$ of T, and define $s \simeq t$ to hold iff $\mathcal{M} \models s = t$. The relation $\simeq$ is a c-model of T. □

3.3.9 Corollary. *For $q = \mathrm{p}, \mathrm{e}, \mathrm{c}$, a quantifier-free theory T in a functional language $\mathcal{L}$ and a quantifier-free sentence A, if $T \models_q A$, then $T \models A$.*

Proof. Theorems 3.3.7 and 3.3.8 imply that for all quantifier-free theories T and sentences A the following statements are equivalent: $T, \mathrm{E}^{\mathcal{L}}, \mathrm{C}^{\mathcal{L}} \models_{\mathrm{p}} A$; $T, \mathrm{C}^{\mathcal{L}} \models_{\mathrm{e}} A$; $T \models_{\mathrm{c}} A$; $T \models A$. The corollary then follows from the straightforward fact that if $T \models_q A$ and $S \supseteq T$, then $S \models_q A$. □

3.3.10 Induced equivalences. Let D be a domain, and T a set of sentences. We define the *equivalence over D induced by T*, denoted by $\simeq_T^D$, to be such that $s \simeq_T^D t$ holds iff $T \models_{\mathrm{e}} s = t$. We will drop the superscript when the domain is known from the context. In the following chapters, we will often use equivalences induced by the set of closed equalities occurring in a set of sentences T. We will denote this set by T^{e}, i.e., $T^{\mathrm{e}} = \{\, s = t \mid (s = t) \in T \,\}$. We will also occasionally need the following technical lemma about sets of closed equalities.

3.3.11 Lemma. *Let Γ be a set of equalities in a language $\mathcal{L}$, and let s_1, s_2, t_1, t_2 be terms of that language. Then $s_1 = t_1, \Gamma \models_{\mathrm{e}} s_2 = t_2$ iff (i) $\Gamma \models_{\mathrm{e}} s_2 = t_2$, or (ii) $\Gamma \models_{\mathrm{e}} s_1 = s_2, t_1 = t_2$, or (iii) $\Gamma \models_{\mathrm{e}} s_1 = t_2, t_1 = s_2$.*

Proof. The direction ($\leftarrow$) is a straightforward consequence of the properties of equivalence in all three cases. For the direction ($\rightarrow$), observe:

(a) For any term t, we have $s_1 = t_1, \Gamma \models_{\mathrm{e}} s_1 = t$ iff $s_1 = t_1, \Gamma \models_{\mathrm{e}} t_1 = t$ iff $\Gamma \models_{\mathrm{e}} s_1 = t$ or $\Gamma \models_{\mathrm{e}} t_1 = t$.

(b) For any s, t such that $\Gamma \models_{\mathrm{e}} s = t$, we trivially have $s_1 = t_1, \Gamma \models_{\mathrm{e}} s = t$.

(c) If t is a term such that $\Gamma \not\models_{\mathrm{e}} s_1 = t$ and $\Gamma \not\models_{\mathrm{e}} t_1 = t$, then for any term s, $\Gamma \models_{\mathrm{e}} s = t$ iff $s_1 = t_1, \Gamma \models_{\mathrm{e}} s = t$.

Armed with these simple observations, assume $s_1 = t_1, \Gamma \vDash_e s_2 = t_2$ and consider two possibilities: If $\Gamma \vDash_e s_1 = s_2$, then also $s_1 = t_1, \Gamma \vDash_e s_1 = s_2$ by (b), then $s_1 = t_1, \Gamma \vDash_e t_1 = s_2$ by (a), hence $s_1 = t_1, \Gamma \vDash_e t_1 = t_2$ by the assumption and transitivity, thus $\Gamma \vDash_e s_1 = t_2$ or $\Gamma \vDash_e t_1 = t_2$ by (a) again. In the first case, (i) holds; in the second case, (ii) holds. If, on the other hand, $\Gamma \nvDash_e s_1 = s_2$, consider two subcases: If $\Gamma \vDash_e t_1 = s_2$, using analogous reasoning as above we obtain $\Gamma \vDash_e s_1 = t_2$ or $\Gamma \vDash_e t_1 = t_2$, whence (iii) or (i). Finally, if $\Gamma \nvDash_e t_1 = s_2$, then (i) holds by (c). □

3.4 Congruence Proof System

3.4.1 States of the congruence closure algorithm. Recall from Par. 3.1.1 that the goal of the congruence closure algorithm is to decide $\Gamma_0 \vDash s_0 = t_0$ for a set of equalities Γ_0 and terms s_0, t_0. During the run of the algorithm, the members of Γ_0 are considered sequentially. With each equality, the equivalence-representing data structure is updated, and immediate consequences of the congruence property 3.3.1(iv) are added to the set of equalities to be considered.

We wish to define a formal proof system that will reflect the behavior of the algorithm. Towards that end, we need to describe the state of the algorithm at a given point in time. Let Δ denote the set of equalities already considered, and let Π denote the set of equalities to be considered (pending). Note that both Δ and Π may contain equalities derived from congruence. The invariant of the algorithm is as follows: For all $s, t \in D$ (D is the domain of $\Gamma_0, s_0 = t_0$), $\Gamma_0 \vDash s = t$ is equivalent to $\Pi, \Delta \vDash s = t$.

The equalities in Δ are reflected in the equivalence-representing data structure. Namely, the data structure represents the equivalence $\simeq_\Delta^D$ as defined in Par. 3.3.10. Using this data structure, it is possible to immediately decide $\Delta \vDash_e s = t$ for any $s, t \in D$.

In the formal system defined in the next paragraph, we will join the sets Δ and Π into a single set $\Gamma = \Delta, \{ \Rightarrow s = t \mid (s = t) \in \Pi \}$. We will have $\Delta = \Gamma^e$, where the notation Γ^e was defined in Par. 3.3.10. The pending equalities Π are marked syntactically as atomic clauses $\Gamma \setminus \Gamma^e$. Semantically, the latter set is equivalent to Π.

3.4.2 The C proof system. Assume for the rest of this section that $\mathcal{L}$ is a functional language, D is a finite domain in $\mathcal{L}$, all sets of formulas are D-bounded. Hence the following definitions are parameterized by D. Finite sets of formulas containing only equalities and atomic clauses (similar to the set Γ from the previous paragraph) will be referred to as *atomic theories*. The *C proof system* is a formal system consisting of derivation rules

$$\frac{(\Rightarrow f(\vec{s}) = f(\vec{t})), \Gamma}{\Gamma} \qquad \text{if } \Gamma^e \vDash_e \vec{s} = \vec{t}, \text{ and } f(\vec{s}), f(\vec{t}) \in D; \tag{1}$$

$$\frac{(s = t), \Gamma}{\Gamma} \qquad \text{if } (\Rightarrow s = t) \in \Gamma; \tag{2}$$

where Γ ranges over all atomic theories. The derivation rule (1) corresponds to adding an immediate consequence of the congruence property to the set of pending equalities. The semantic condition $\Gamma^{\mathrm{e}} \models_{\mathrm{e}} \vec{s} = \vec{t}$ the first rule is rather non-standard. However, this condition is decidable using an equivalence-representing data structure, such as the one from [Tar75]. The derivation rule (2) corresponds to considering a pending equality, and updating the equivalence-representing data structure.

A *C derivation* is a finite sequence $\Gamma_1, \ldots, \Gamma_n$ of atomic theories such that for each i, $1 \leq i < n$, Γ_{i+1} is the conclusion of some derivation rule of the C proof system for the premise Γ_i. Note that for each $1 \leq i < n$, $\Gamma_1 \subseteq \Gamma_i \subseteq \Gamma_{i+1}$, hence for any formula A, we have $\Gamma_{i+1} \models A$ whenever $\Gamma_i \models A$. We say that an equality $s = t$ between terms $s, t \in D$ is *C-derivable* from an atomic theory Γ, and write $\Gamma \vdash_{\mathrm{C}} s = t$, if there is a *C derivation of $s = t$ from Γ*, i.e., a C derivation starting with Γ such that in its final element Γ', we have $\Gamma'^{\mathrm{e}} \models_{\mathrm{e}} s = t$.

We wish to prove that the C proof system is sound and complete, i.e., that C derivability is equivalent to logical consequence from atomic theories for equalities among terms in D (Thm. 3.4.6). Towards this end, we need the following definitions and lemmas.

3.4.3 (Weak) C saturation. An atomic theory Γ is called *saturated with respect to a rule of derivation* of the C proof system if one of the conclusions of every possible application of the rule to Γ is Γ itself. There are no derivation rules with multiple conclusions in the C proof system. The definition is deliberately more general so as to be usable in later chapters, where there will be such derivation rules.

An atomic theory Γ is *C-saturated* if it is saturated w.r.t. both rules of the C proof system, and it is *weakly C-saturated* if it is saturated w.r.t. 3.4.2(1), and we have $\Gamma^{\mathrm{e}} \models_{\mathrm{e}} s = t$ for every atomic clause $(\Rightarrow s = t) \in \Gamma$. A C-saturated atomic theory is obviously weakly C-saturated. Notice also that unlike c-closure of binary relations on terms in Par. 3.3.4, saturation is a syntactic property.

Lemma 3.4.4(i) confirms that neither rule of the C proof system derives a conclusion that is not a logical consequence of its premise. According to Lemma 3.4.4(ii), we can C-saturate any atomic theory. Lemma 3.4.5 states that the subset Γ^{e} of any weakly C-saturated atomic theory Γ contains enough consequences of the congruence property so that any e-model of Γ^{e} possesses all properties of equality for terms of D.

3.4.4 Lemma. *Let Γ be an atomic theory.*

(i) If Γ' is a conclusion of an application of a C derivation rule to Γ, then Γ' is a conservative extension of Γ for equalities among the terms of D.

(ii) There exists a C derivation starting with Γ and ending with a C-saturated atomic theory.

Proof. (i) First, Γ' is an extension of Γ, since $\Gamma \subseteq \Gamma'$, so $\Gamma' \models \Gamma$. For conservativity of Γ' over Γ, consider two cases: If Γ' is a conclusion of the rule 3.4.2(1), then there are $f(\vec{s}), f(\vec{t}) \in D$ such that $\Gamma^{\mathrm{e}} \models_{\mathrm{e}} \vec{s} = \vec{t}$, and $\Gamma' = (\Rightarrow f(\vec{s}) = f(\vec{t})), \Gamma$. Since

$\Gamma^{\mathrm{e}} \subseteq \Gamma$, and every congruence is an equivalence, we also have $\Gamma \vDash_{\mathrm{c}} \vec{s} = \vec{t}$, whence $\Gamma \vDash_{\mathrm{c}} f(\vec{s}) = f(\vec{t})$, thus $\Gamma \vDash_{\mathrm{c}} \Gamma'$, which implies $\Gamma \vDash \Gamma'$ by Thm. 3.3.8(ii). Therefore for any equality $s = t$ among terms of D, if $\Gamma' \vDash s = t$, then $\Gamma \vDash s = t$. In case Γ' is a conclusion of 3.4.2(1), there is $(\Rightarrow u = v) \in \Gamma$, and $\Gamma' = (u = v), \Gamma$. Now clearly every model of Γ is a model of Γ', therefore again $\Gamma \vDash \Gamma'$, and conservativity follows.

(ii) Note that every atomic theory $\Gamma \subseteq \Gamma_D$, where $\Gamma_D = \bigcup_{s,t \in D} \{s = t, (\Rightarrow s = t)\}$, which is a finite set, since D is finite. The lemma is easily proved by induction on k for all Γ such that $|\Gamma_D \setminus \Gamma| = k$. For $k = 0$, we have $\Gamma = \Gamma_D$. Conclusion of neither rule of the C proof system can be a proper superset of Γ_D, thus the set is C-saturated, and the single-element sequence Γ_D is the C derivation searched for. Take $k \geq 0$ and any Γ such that $|\Gamma_D \setminus \Gamma| = k + 1$. If Γ is not C-saturated, then at least one of the rules of the C proof system can be applied to Γ with a conclusion Γ' a proper superset of Γ. For either rule, $|\Gamma_D \setminus \Gamma'| = k$. We obtain the required C derivation by prepending Γ to a C derivation starting with Γ' and ending with a C-saturated set, which exists by the induction hypothesis. □

3.4.5 Lemma. *Let Γ be a weakly C-saturated atomic theory. For all terms $s, t \in D$, we have $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = t$ iff $\Gamma \vDash_{\mathrm{c}} s = t$.*

Proof. Take any weakly C-saturated atomic theory Γ, and any terms $s, t \in D$. For the direction ($\rightarrow$), assume $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = t$. Any congruence $\simeq$ such that $\simeq \vDash \Gamma$ is an equivalence, and $\simeq \vDash \Gamma^{\mathrm{e}}$ since $\Gamma^{\mathrm{e}} \subseteq \Gamma$. Hence $\simeq \vDash s = t$ by the assumption.

For the direction ($\leftarrow$), we assume $\Gamma^{\mathrm{e}} \nvDash_{\mathrm{e}} s = t$, and prove $\Gamma \nvDash_{\mathrm{c}} s = t$: Let us construct the equivalence $\simeq_{\Gamma^{\mathrm{e}}}$ over D induced by Γ^{e} (Par. 3.3.10). For any $f(\vec{s}), f(\vec{t}) \in D$, if $\vec{s} \simeq_{\Gamma^{\mathrm{e}}} \vec{t}$, then $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{s} = \vec{t}$ by the definition of induced equivalence, and we get $(\Rightarrow f(\vec{s}) = f(\vec{t})) \in \Gamma$ and $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} f(\vec{s}) = f(\vec{t})$ from weak C saturation of Γ, so $f(\vec{s}) \simeq_{\Gamma^{\mathrm{e}}} f(\vec{t})$. The equivalence $\simeq_{\Gamma^{\mathrm{e}}}$ is thus c-closed over D. By Lemma 3.3.5, it can be expanded to a congruence $\simeq$ over $D_{\mathcal{L}}$ coinciding with $\simeq_{\Gamma^{\mathrm{e}}}$ on D. Hence $\simeq \vDash \Gamma^{\mathrm{e}}$. All formulas $A \in \Gamma \setminus \Gamma^{\mathrm{e}}$ are atomic clauses. Whenever $(\Rightarrow u = v) \in \Gamma$ for some terms $u, v \in D$, then $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} u = v$ since Γ is weakly C-saturated, consequently $u \simeq_{\Gamma^{\mathrm{e}}} v$, and $u \simeq v$ by coincidence. Therefore $\simeq \vDash \Gamma$. But at the same time, we have $\simeq \nvDash s = t$ by coincidence. Thus $\Gamma \nvDash_{\mathrm{c}} s = t$. □

3.4.6 Theorem (Soundness and completeness of the C proof system). *Let Γ be an atomic theory. For all terms $s, t \in D$, we have $\Gamma \vdash_{\mathrm{C}} s = t$ iff $\Gamma \vDash s = t$.*

Proof. To prove soundness ($\rightarrow$), assume that $\Gamma \vdash_{\mathrm{C}} s = t$, i.e., that there is a C derivation $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$, with $\Gamma_1 = \Gamma$, and $\Gamma_n^{\mathrm{e}} \vDash_{\mathrm{e}} s = t$. We show by induction on i that if $1 \leq i \leq n$, and $\Gamma_i^{\mathrm{e}} \vDash_{\mathrm{e}} s = t$, then $\Gamma \vDash s = t$. For $i = 1$, we have $\Gamma \vDash_{\mathrm{e}} s = t$, and consequently $\Gamma \vDash_{\mathrm{c}} s = t$, thus $\Gamma \vDash s = t$ by Thm. 3.3.8(ii). For $i \geq 1$, $\Gamma_i \vDash \Gamma_{i+1}$ by conservativity Lemma 3.4.4(i), and by the induction hypothesis, we have $\Gamma_i \vDash s = t$, hence $\Gamma_{i+1} \vDash s = t$.

For completeness ($\leftarrow$), assume $\Gamma \nvdash_{\mathrm{C}} s = t$. By Lemma 3.4.4(ii), there is a C derivation starting with Γ and ending with some C-saturated theory Γ'. We have $\Gamma'^{\mathrm{e}} \nvDash_{\mathrm{e}} s = t$ by the assumption. As noted in Par. 3.4.3, a C-saturated theory is weakly C-saturated, therefore $\Gamma' \nvDash_{\mathrm{c}} s = t$ by Lemma 3.4.5. Thus there is a congruence $\simeq$ satisfying Γ', but not satisfying $s = t$. Since $\Gamma \subseteq \Gamma'$ (Par. 3.4.2), $\simeq \vDash \Gamma$, therefore $\Gamma \nvDash_{\mathrm{c}} s = t$. Consequently $\Gamma \nvDash s = t$ by Thm. 3.3.8(ii). □

3.4.7 Auxiliary properties of C derivations. In the next section, we will need the following definitions and a couple of technical properties of C derivation summarized in Lemma 3.4.8. A C derivation $\Gamma_1, \ldots, \Gamma_n$ is *strictly monotonic* if $\Gamma_i \subsetneq \Gamma_{i+1}$ for each $1 \leq i < n$. For every atomic theory Γ, let us denote by $\mathrm{C}(\Gamma)$ the set $\{\, (\Rightarrow f(\vec{s}) = f(\vec{t})) \mid f(\vec{s}), f(\vec{t}) \in D \text{ and } \Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{s} = \vec{t} \,\}$.

3.4.8 Lemma.

(i) Let Γ be an atomic theory. There exists a strictly monotonic C derivation starting with Γ, applying only the rule 3.4.2(1), and such that its final element is the 3.4.2(1)-saturated set $\mathrm{C}(\Gamma), \Gamma$.

(ii) Let Δ and Π be sets of equalities such that $\Pi \vDash_{\mathrm{e}} \Delta$. Let $\Gamma_1, \ldots, \Gamma_n$ be such a C derivation that $\Delta \subseteq \Gamma_1$. Let the derivation be strictly monotonic if $\Delta \neq \emptyset$. Then the sequence $\Gamma'_1, \ldots, \Gamma'_n$, where $\Gamma'_i = (\Gamma_i \setminus \Delta) \cup \Pi$ for all $1 \leq i \leq n$, is a C derivation.

Proof. (i) If $n = 0$, then $\mathrm{C}(\Gamma) \subseteq \Gamma$. Hence Γ is 3.4.2(1)-saturated, and a single-element sequence Γ satisfies the claim. Assume the claim holds for $n \geq 0$, and take any Γ such that $|\mathrm{C}(\Gamma) \setminus \Gamma| = n + 1$. Then there are $f(\vec{s}), f(\vec{t}) \in D$ such that $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{s} = \vec{t}$, and $(\Rightarrow f(\vec{s}) = f(\vec{t})) \notin \Gamma$. Therefore the rule 3.4.2(1) is applicable. Let us denote its conclusion $(\Rightarrow f(\vec{s}) = f(\vec{t})), \Gamma$ by Γ'. Clearly $\Gamma \subsetneq \Gamma'$, and $|C(\Gamma') \setminus \Gamma'| = n$ since $\Gamma^{\mathrm{e}} = \Gamma'^{\mathrm{e}}$. Thus by the induction hypothesis, there is a strictly monotonic C derivation starting with Γ', applying only 3.4.2(1), and such that its final element is $\mathrm{C}(\Gamma'), \Gamma' = \mathrm{C}(\Gamma), \Gamma$. By prepending Γ to the C derivation, we obtain C derivation satisfying the claim.

(ii) Take any Γ, Δ, Π, and $\Gamma_1, \ldots, \Gamma_n$ satisfying the assumptions. Let us construct the sequence $\Gamma'_1, \ldots, \Gamma'_n$ as required. To prove that the sequence is a C derivation, take any $1 \leq i < n$, and consider two cases: If the rule 3.4.2(1) was applied to Γ_i to form Γ_{i+1}, then there are $f(\vec{s}), f(\vec{t}) \in D$ such that $\Gamma_i^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{s} = \vec{t}$ and $\Gamma_{i+1} = \{\Rightarrow f(\vec{s}) = f(\vec{t})\} \cup \Gamma_i$. As in all C derivations, $\Gamma_1 \subseteq \Gamma_i$, hence $\Delta \subseteq \Gamma_i$. By the assumption $\Pi \vDash_{\mathrm{e}} \Delta$, every equivalence satisfying Π satisfies Δ, whence we get $(\Gamma'_i)^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{s} = \vec{t}$. Therefore the rule can be applied to Γ'_i. Since Δ contains only equalities, $(\Rightarrow f(\vec{s}) = f(\vec{t})) \notin \Delta$, and consequently $\Gamma'_{i+1} = (\Gamma_{i+1} \setminus \Delta) \cup \Pi = \{\Rightarrow f(\vec{s}) = f(\vec{t})\} \cup (\Gamma_i \setminus \Delta) \cup \Pi = \{\Rightarrow f(\vec{s}) = f(\vec{t})\} \cup \Gamma'_i$. Thus Γ'_{i+1} is indeed the conclusion of the application of 3.4.2(1) to Γ'_i for $f(\vec{s})$ and $f(\vec{t})$.

If the derivation rule 3.4.2(2) was applied to Γ_i to obtain Γ_{i+1}, then there is $(\Rightarrow s = t) \in \Gamma_i$, and $\Gamma_{i+1} = \{s = t\} \cup \Gamma_i$. Since there are no atomic clauses in Δ, we have $(\Rightarrow s = t) \in (\Gamma_i \setminus \Delta) \cup \Pi$. Therefore the derivation rule is also applicable to Γ'_i. In case $\Delta = \emptyset$, we obviously have $(s = t) \notin \Delta$. Otherwise we get $(s = t) \notin \Gamma_i$

from strict monotonicity of $\Gamma_1, \ldots, \Gamma_n$, hence $(s = t) \notin \Delta$ since $\Delta \subseteq \Gamma_1 \subseteq \Gamma_i$. Thus $\Gamma'_{i+1} = \{s = t\} \cup (\Gamma_i \smallsetminus \Delta) \cup \Pi = \{s = t\} \cup \Gamma'_i$ is actually the conclusion of the application of 3.4.2(2) to Γ'_i for $(\Rightarrow s = t)$. □

3.5 A Set-Theoretical Algorithm for Congruence Closure

After establishing a logical framework for the congruence closure in the previous section, we describe the algorithm for it in abstract, set-theoretical terms of equivalences, equivalence classes, and iteratively built set-theoretical trees labeled with equivalence classes.

3.5.1 Equivalence trees. For an equivalence $\simeq$ over a domain D, we define

$$T(\simeq) = \{\, \langle [f(\vec{s})], f, [s_1], \ldots, [s_n], \emptyset \rangle \mid f(\vec{s}) \in D \,\}.$$

We designate by $Tr(\simeq)$ the closure of $T(\simeq)$ under the initial segments of its sequences, i.e., the least set T such that $T(\simeq) \subseteq T$, and with every $\langle e \,|\, p \rangle \in T$ also $p \in T$. The reader will observe that the set $Tr(\simeq)$ is a *tree* in the set-theoretical sense. The finite sequences $p \in Tr(\simeq)$ are its *nodes*. Every node $\langle e \,|\, p \rangle \in Tr(\simeq)$ is a *child* of p, and p is the *parent* of $\langle e \,|\, p \rangle$. The nodes $\langle e_1 \,|\, p \rangle, \langle e_2 \,|\, p \rangle \in Tr(\simeq)$ are *siblings*. The node $\langle \emptyset \rangle$ is the *root*, and nodes without children are *leaves*. We call e the *label* of the node $\langle e \,|\, p \rangle$. The label of the root $\langle \emptyset \rangle$ is $\emptyset$. The tree $Tr(\simeq)$ is finitely branching (each node has finitely many children) with every *path* finite. In our particular case, each path is a leaf and vice versa. The above definitions make $T(\simeq)$ the set of paths through the tree $Tr(\simeq)$, as well as the set of leaves of the tree. We also have $D/{\simeq} = \{\, e \mid \exists p \, \langle e \,|\, p \rangle \in T(\simeq) \,\}$.

3.5.2 Example (Equivalence trees and C derivation rules). Consider the domain $D = \{a, b, c, f(a,b), f(b,c), f(c,b)\}$, and the atomic theory $\Gamma_1 = \{(\Rightarrow a = c), (\Rightarrow b = c)\}$. Fig. 3.1(a) depicts a graphical representation of the equivalence tree $Tr(\simeq_{\Gamma_1^{\mathrm{e}}})$ for $\simeq_{\Gamma_1^{\mathrm{e}}}$, which is an identity on D since $\Gamma_1^{\mathrm{e}} = \emptyset$. By applying the derivation rule 3.4.2(2) to Γ_1 and $(\Rightarrow a = c)$, we obtain the atomic theory $\Gamma_2 = \{a = c, (\Rightarrow a = c), (\Rightarrow b = c)\}$.

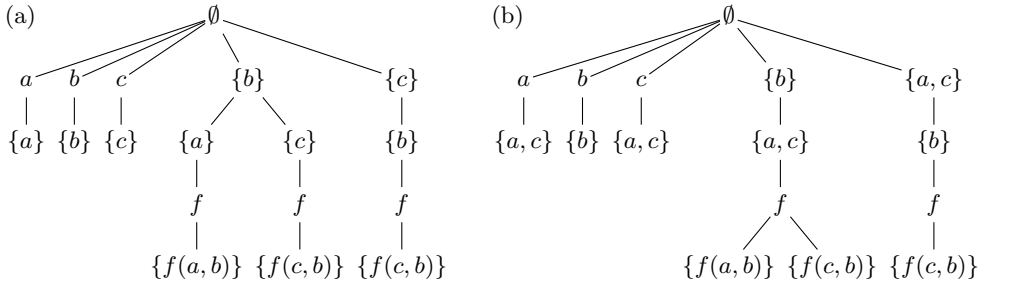


Figure 3.1. Examples of equivalence trees.

The tree $Tr(\simeq_{\Gamma_2^e})$ is shown in Fig. 3.1(b). Notice that there are now two sibling leaves in the tree, labeled $\{f(a,b)\}$ and $\{f(c,b)\}$ respectively, because their respective ancestors labeled $\{a\}$ and $\{c\}$ in $Tr(\simeq_{\Gamma_1^e})$ have been merged into a common ancestor labeled $\{a,c\}$ in $Tr(\simeq_{\Gamma_2^e})$. This is an indication that the rule 3.4.2(1) can be applied to Γ_2, $f(a,b)$, and $f(c,b)$, since $\Gamma_2^e \vDash_e a = c, b = b$.

Taking the example a step further, we apply the rule indicated above, and obtain $\Gamma_3 = \{(\Rightarrow f(a,b) = f(c,b)), a = c, (\Rightarrow a = c), (\Rightarrow b = c)\}$. Now Γ_3 is saturated w.r.t. 3.4.2(1), however $T(\simeq_{\Gamma_3^e}) = T(\simeq_{\Gamma_2^e})$ since $\Gamma_3^e = \Gamma_2^e$. The presence of sibling leaves in trees for atomic theories saturated w.r.t. 3.4.2(1) can be regarded as an indication that the rule 3.4.2(2) was not yet applied to conclusions of 3.4.2(1) [$(\Rightarrow f(a,b) = f(c,b))$ in our example].

3.5.3 Trees for atomic theories. We can represent all atomic clauses from an atomic theory Γ in a tree by defining

$$T(\Gamma) = T(\simeq_{\Gamma^e}) \cup \{\, \langle [s], f, [t_1], \ldots, [t_n], \emptyset \rangle \mid (\Rightarrow s = f(\vec{t})) \in \Gamma \,\}.$$

$Tr(\Gamma)$ is defined analogously to $Tr(\simeq)$. We also extend the notions of root, parent, sibling, etc. defined in Par. 3.5.1 for equivalence trees to trees for atomic theories.

The tree $T(\Gamma_1)$ for the theory Γ_1 from Par. 3.5.2 is shown in Fig. 3.2(a). We again have $T(\Gamma_2) = T(\Gamma_3)$, and this tree is displayed in Fig. 3.2(b). Not yet considered atomic clauses $(\Rightarrow b = c)$ and $(\Rightarrow f(a,b) = f(c,b))$ in 3.4.2(1)-saturated set Γ_3 are represented in the tree $T(\Gamma_3)$ by two pairs of sibling leaves: the pair labeled $\{a,c\}$ and $\{b\}$ respectively, and the pair labeled $\{f(a,b)\}$ and $\{f(c,b)\}$ respectively. A similar relationship can be found between sibling leaves in $T(\Gamma_1)$ on one hand, and not yet considered atomic clauses in Γ_1 (which is trivially 3.4.2(1)-saturated) on the other hand.

The following lemma summarizes some properties of trees for atomic theories. The first part makes explicit the link between the labels of leaves in a tree for an atomic theory and e-consequence of the set of equalities of that theory. The parts (ii) and (iii) state that trees are invariant under addition of e-consequences, and under application of the rule 3.4.2(1) respectively. The part (iii) thus generalizes

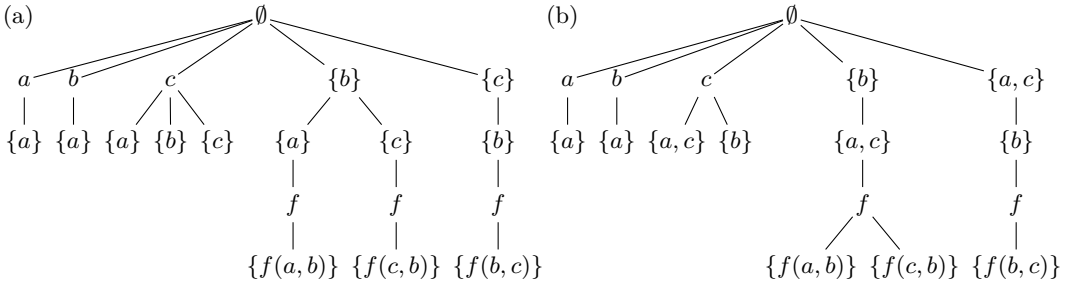


Figure 3.2. Examples of trees for atomic theories.

our observation that $T(\Gamma_2) = T(\Gamma_3)$ in the example. The part (iv) hints, and the part (v) explains precisely the meaning of presence of sibling leaves in a tree, thus generalizing our observation on sibling leaves in $T(\Gamma_1)$ and $T(\Gamma_3)$.

3.5.4 Lemma. *Let Γ be any atomic theory.*

(i) For all terms $s, t \in D$, there is a leaf $\langle e \,|\, r\rangle \in T(\Gamma)$ such that $s, t \in e$ iff $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = t$.

(ii) For all terms $s, t \in D$, if $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = t$, then $T((s = t), \Gamma) = T(\Gamma)$.

(iii) If Γ' is an atomic theory such that $\Gamma \subseteq \Gamma' \subseteq \Gamma \cup \mathrm{C}(\Gamma)$, then $T(\Gamma') = T(\Gamma)$.

(iv) If $\Gamma^{\mathrm{e}} \nvDash_{\mathrm{e}} s = t$ for some $(\Rightarrow s = t) \in \Gamma \cup \mathrm{C}(\Gamma)$, then there is a pair of sibling leaves with distinct labels in $T(\Gamma)$.

(v) Let e, e' be any sets, $e \neq e'$. There is a pair of sibling leaves in $T(\Gamma)$ labeled by e and e' respectively iff there are terms $s, t, f(\vec{s}), f(\vec{t}) \in D$ such that $(\Rightarrow s = f(\vec{s}))$, $(\Rightarrow t = f(\vec{t})), (\Rightarrow f(\vec{s}) = f(\vec{t})) \in \Gamma \cup \mathrm{C}(\Gamma)$, $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{s} = \vec{t}$, $e = [s]$, and $e' = [t]$.

Proof. (i) Take any atomic theory Γ, and arbitrary terms $s, t \in D$. If $\langle e \,|\, r\rangle \in T(\Gamma)$ and $s, t \in e$, then, regardless of whether $\langle e \,|\, r\rangle \in T(\simeq_{\Gamma^{\mathrm{e}}})$ or $\langle e \,|\, r\rangle \in T(\Gamma) \setminus T(\simeq_{\Gamma^{\mathrm{e}}})$, we have $s \simeq_{\Gamma^{\mathrm{e}}} t$, thus $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = t$. Conversely, if $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = t$, then there is a function symbol f of arity $n \geq 0$ and terms $\vec{s}$ such that $s \equiv f(\vec{s})$. By the definition of $T(\simeq_{\Gamma^{\mathrm{e}}})$, there is a leaf $\langle [s], f, [s_1], \ldots, [s_n], \emptyset\rangle \in T(\simeq_{\Gamma^{\mathrm{e}}}) \subseteq T(\Gamma)$, and $t \in [s]$, since $s \simeq_{\Gamma^{\mathrm{e}}} t$.

(ii) Take any atomic theory Γ, any terms $s, t \in D$, and assume $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = t$. Hence each equivalence satisfying Γ^{e} satisfies $(s = t), \Gamma^{\mathrm{e}}$, therefore for every $u, v \in D$, we have $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} u = v$ iff $(s = t), \Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} u = v$, whence $\simeq_{\Gamma^{\mathrm{e}}} = \simeq_{(s=t),\Gamma^{\mathrm{e}}}$, and the respective equivalence trees are equal. Also the set of atomic clauses in Γ is equal to the set of atomic clauses in $(s = t), \Gamma$. Therefore $T(\Gamma) = T(s = t, \Gamma)$.

(iii) Take any atomic theories Γ and Γ' satisfying the assumption. Since $(\mathrm{C}(\Gamma))^{\mathrm{e}}$ is empty, we have $\Gamma'^{\mathrm{e}} = \Gamma^{\mathrm{e}}$, thus $T(\simeq_{\Gamma'^{\mathrm{e}}}) = T(\simeq_{\Gamma^{\mathrm{e}}})$. As $\Gamma \subseteq \Gamma'$, we have $T(\Gamma) \setminus T(\simeq_{\Gamma^{\mathrm{e}}}) \subseteq T(\Gamma') \setminus T(\simeq_{\Gamma'^{\mathrm{e}}})$ by the definition, whence $T(\Gamma) \subseteq T(\Gamma')$. To prove $T(\Gamma') \subseteq T(\Gamma)$, it is sufficient to show $T(\Gamma') \setminus T(\simeq_{\Gamma'^{\mathrm{e}}}) \subseteq T(\Gamma)$. Take any $p \in T(\Gamma') \setminus T(\simeq_{\Gamma'^{\mathrm{e}}})$. By the definition, there is $(\Rightarrow s = f(\vec{t})) \in \Gamma'$ such that $p = \langle [s], f, [t_1], \ldots, [t_n], \emptyset\rangle$. If $(\Rightarrow s = f(\vec{t})) \in \Gamma$, then obviously $p \in T(\Gamma)$. Otherwise $(\Rightarrow s = f(\vec{t})) \in \Gamma' \setminus \Gamma \subseteq \mathrm{C}(\Gamma)$. By the definition of $\mathrm{C}(\Gamma)$, there is $f(\vec{s}) \in D$ such that $s \equiv f(\vec{s})$, and $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{s} = \vec{t}$. But then $[s_i] = [t_i]$ for all $1 \leq i \leq n$, thus $p = \langle [f(\vec{s})], f, [s_1], \ldots, [s_n], \emptyset\rangle \in T(\simeq_{\Gamma^{\mathrm{e}}}) \subseteq T(\Gamma)$.

(iv) Take any atomic theory Γ, and any $(\Rightarrow s = t) \in \Gamma \cup \mathrm{C}(\Gamma)$ such that $\Gamma^{\mathrm{e}} \nvDash_{\mathrm{e}} s = t$. Assume $(\Rightarrow s = t) \in \Gamma$. Since there is a function symbol f and terms $\vec{t} \in D$ such that $t \equiv f(\vec{t})$, there is a leaf $p := \langle [t], f, [t_1], \ldots, [t_n], \emptyset\rangle \in T(\simeq_{\Gamma^{\mathrm{e}}}) \subseteq T(\Gamma)$. There also is $q := \langle [s], f, [t_1], \ldots, [t_n], \emptyset\rangle \in T(\Gamma)$. The leaves p and q are clearly siblings, and since $\Gamma^{\mathrm{e}} \nvDash_{\mathrm{e}} s = t$, their respective labels $[s]$ and $[t]$ are distinct. For the second case, assume $(\Rightarrow s = t) \in \mathrm{C}(\Gamma)$. Then there are $f(\vec{s}), f(\vec{t}) \in D$ such that $s \equiv f(\vec{s})$, $t \equiv f(\vec{t})$, $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{s} = \vec{t}$. Thus there are leaves $\langle [f(\vec{s})], f, [s_1], \ldots, [s_n], \emptyset\rangle$, $\langle [f(\vec{t})], f, [t_1], \ldots, [t_n], \emptyset\rangle \in T(\simeq_{\Gamma^{\mathrm{e}}}) \subseteq T(\Gamma)$. The leaves are siblings since $[s_i] = [t_i]$ for all $i = 1, \ldots, n$ by the assumption $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{s} = \vec{t}$. The respective labels $[f(\vec{s})]$ and $[f(\vec{t})]$ of these sibling leaves are distinct since $\Gamma^{\mathrm{e}} \nvDash_{\mathrm{e}} f(\vec{s}) = f(\vec{t})$.

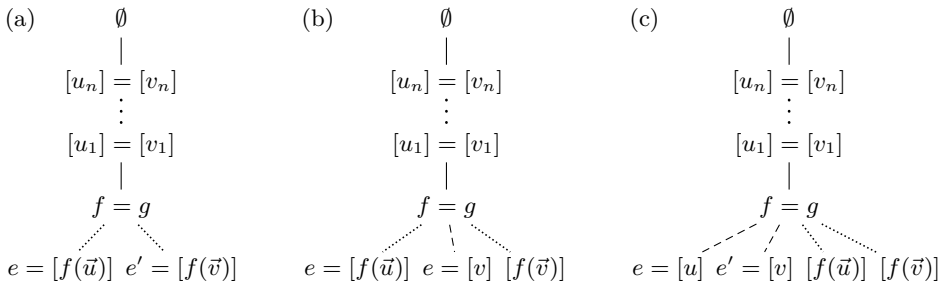


Figure 3.3. An illustration of cases in the proof of Lemma 3.5.4(v).

(v) The direction ($\leftarrow$) is similar to (iv): Take any Γ, e, e', s, t, $f(\vec{s})$, and $f(\vec{t})$ satisfying the assumption. We have $\langle [s], f, [s_1], \ldots, [s_n], \emptyset\rangle, \langle [t], f, [t_1], \ldots, [t_n], \emptyset\rangle \in T(\Gamma)$, and these leaves are siblings (since $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{s} = \vec{t}$) with distinct labels (since $[s] = e \neq e' = [t]$). For the direction ($\rightarrow$), take any atomic theory Γ and sets $e \neq e'$, and assume that in $T(\Gamma)$ there are siblings $p := \langle e \,|\, r\rangle$ and $q := \langle e' \,|\, r\rangle$ for some r. If $p \in T(\simeq_{\Gamma^{\mathrm{e}}})$, then for some $f(\vec{u}) \in D$, we have $e = [f(\vec{u})]$, and $r = \langle f, [u_1], \ldots, [u_m], \emptyset\rangle$. If also $q \in T(\simeq_{\Gamma^{\mathrm{e}}})$, then there is $g(\vec{v}) \in D$ such that $e' = [g(\vec{v})]$, and $r = \langle g, [v_1], \ldots, [v_n], \emptyset\rangle$. Obviously, we have $f = g$, $m = n$, and $u_i \simeq_{\Gamma^{\mathrm{e}}} v_i$, $i = 1, \ldots, n$ (cf. Fig. 3.3(a)), thus $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{u} = \vec{v}$. From the definition of $\mathrm{C}(\Gamma)$, we get $(\Rightarrow f(\vec{u}) = f(\vec{v})), (\Rightarrow f(\vec{u}) = f(\vec{u})), (\Rightarrow f(\vec{v}) = f(\vec{v})) \in \mathrm{C}(\Gamma)$. If $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} f(\vec{u}) = f(\vec{v})$ held true, we would have a contradiction $e = [f(\vec{u})] = [f(\vec{v})] = e'$. Hence the claim holds for $f := f$, $s := f(\vec{u})$, $t := f(\vec{v})$, $\vec{s} := \vec{u}$, and $\vec{t} := \vec{v}$.

If $q \in T(\Gamma) \setminus T(\simeq_{\Gamma^{\mathrm{e}}})$, and there is $(\Rightarrow v = g(\vec{v})) \in \Gamma$ such that $e' = [v]$, and $r = \langle g, [v_1], \ldots, [v_n], \emptyset\rangle$. Again $f = g$, $m = n$, and we obtain $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{u} = \vec{v}$ (cf. Fig. 3.3(b)), $(\Rightarrow f(\vec{u}) = f(\vec{v})) \in \mathrm{C}(\Gamma)$, and $\Gamma^{\mathrm{e}} \nvDash_{\mathrm{e}} f(\vec{u}) = v$ as above. The claim now holds for $f := f$, $s := f(\vec{u})$, $t := v$, and $\vec{s} := \vec{t} := \vec{v}$. The case when $p \in T(\Gamma) \setminus T(\simeq_{\Gamma^{\mathrm{e}}})$ and $q \in T(\simeq_{\Gamma^{\mathrm{e}}})$ is symmetrical.

Finally, assume that both $p, q \in T(\Gamma) \setminus T(\simeq_{\Gamma^{\mathrm{e}}})$. Now there are $(\Rightarrow u = f(\vec{u}))$, $(\Rightarrow v = g(\vec{v})) \in \Gamma$ such that $e = [u]$, $e' = [v]$, $\langle f, [u_1], \ldots, [u_m], \emptyset\rangle = r = \langle g, [v_1], \ldots, [v_n], \emptyset\rangle$. As in previous cases, we have $f = g$, $m = n$, $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{u} = \vec{v}$ (cf. Fig. 3.3(c)), $(\Rightarrow f(\vec{u}) = f(\vec{v})) \in \mathrm{C}(\Gamma)$, and $\Gamma^{\mathrm{e}} \nvDash_{\mathrm{e}} u = v$. We satisfy the claim by $f := f$, $s := u$, $t := v$, $\vec{s} := \vec{u}$, and $\vec{t} := \vec{v}$. □

3.5.5 Direct transformation of trees for atomic theories. Let us return again to our example. We have extended the atomic theory Γ_1 by adding a new equality $a = c$, obtained the atomic theory Γ_2, and constructed the tree $T(\Gamma_2)$ shown in Fig. 3.2(b). By comparing it to the tree $T(\Gamma_1)$ in Fig. 3.2(a), we see that $T(\Gamma_2)$ can be constructed directly from $T(\Gamma_1)$ by replacing all labels $\{a\}$ and $\{c\}$ by $\{a, c\}$, which is the union of equivalence classes into which the terms a and c belonged

in $\simeq_{\Gamma_1^{\mathrm{e}}}$. We do this replacement because these two classes are merged in $\simeq_{\Gamma_2^{\mathrm{e}}}$, while the other equivalence classes are the same as in $\simeq_{\Gamma_1^{\mathrm{e}}}$.

We define the function Join_classes which does this replacement on any set of paths (leaves) T and sets a, b as

$$\mathrm{Join\_classes}(a, b, T) = \{\, \langle e \star c, f, e_1 \star c, \ldots, e_n \star c, \emptyset\rangle \mid \langle e, f, e_1, \ldots, e_n, \emptyset\rangle \in T \,\}$$

where $c = a \cup b$, and

$$e \star c = \begin{cases} e & \text{if } e \cap c = \emptyset, \\ c & \text{otherwise.} \end{cases}$$

Lemma 3.5.6 confirms that the function indeed behaves as described above.

3.5.6 Lemma. *Let Γ be any atomic theory. For every pair of terms $s, t \in D$, we have $T(s = t, \Gamma) = \mathrm{Join\_classes}([s], [t], T(\Gamma))$.*

Proof. Take any atomic theory Γ, choose arbitrary $s, t \in D$, and let $S = [s]_{\simeq_{\Gamma^{\mathrm{e}}}}$, $T = [t]_{\simeq_{\Gamma^{\mathrm{e}}}}$. Clearly, $(s = t, \Gamma)^{\mathrm{e}} = (s = t), \Gamma^{\mathrm{e}}$. Take any $u \in D$, let $U = [u]_{\simeq_{\Gamma^{\mathrm{e}}}}$, $U' = [u]_{\simeq_{s=t,\Gamma^{\mathrm{e}}}}$, and observe

$$\begin{aligned} U' &= \{\, v \in D \mid s = t, \Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} u = v \,\} \\ &= \{\, v \in D \mid \Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} u = v, \text{ or } \Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = u, t = v, \text{ or } \Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = v, t = u \,\} \\ &= U \cup \{\, v \in D \mid \Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = u, t = v \,\} \cup \{\, v \in D \mid \Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = v, t = u \,\}, \qquad (*) \end{aligned}$$

where the second step is by Lemma 3.3.11. Consider now the following cases: If we have $U \cap (S \cup T) = \emptyset$, then the second and third sets in the union $(*)$ are empty, thus $U' = U$. If $U \cap S \neq \emptyset$, then, since U and S are equivalence classes of $\simeq_{\Gamma^{\mathrm{e}}}$, $S = U$ and $\Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = u$. Thus the second set in the union $(*)$ is equal to T. The third set is either empty (if $U \cap T = \emptyset$), or equal to S (otherwise). In either case, $U' = U \cup T = S \cup T$. The case $U \cap T \neq \emptyset$ analogously leads to $U' = S \cup T$. We thus conclude $U' = U \star (S \cup T)$. The lemma now follows by this and the definition of Join_classes. □

3.5.7 Iteration of Join_classes. We know from Lemma 3.5.4(v) that sibling leaves in a tree indicate the presence of at least one atomic clause in the corresponding 3.4.2(1)-saturated theory. Thus if a, b are the labels of a pair of sibling leaves in $T(\Gamma)$, $\mathrm{Join\_classes}(a, b, T(\Gamma))$ is the tree for the theory which is the conclusion of application of 3.4.2(2) to Γ. (To be precise, this is only true for some pairs of sibling leaves, but we are trying to establish motivation here, and will go into details below.) Moreover, $\mathrm{Join\_classes}(a, b, T(\Gamma))$ is also the tree for the 3.4.2(1)-saturated superset of this theory by Lemma 3.5.4(iii).

Further, if we iteratively apply Join_classes to a tree $T(\Gamma)$, and we end up with a tree T' with no sibling leaves, then all atomic clauses of the theory corresponding to T' have been considered, and the theory is C-saturated. We know from Lemma 3.4.5 that in such theories the set of equalities is sufficient to determine logical

consequence for equalities among the terms of D. Therefore $\Gamma \vDash s = t$ iff we can find a label (equivalence class) in T' to which both s and t belong.

This gives us an algorithm for deciding $\Gamma \vDash s = t$ by iterative transformation of (set-theoretical) trees by the function Join_classes. We formalize the above explanation in the next paragraph, and the lemma and theorem following it.

3.5.8 Tree derivations. For all trees T, T' for atomic theories, we define $T \rightarrowtail T'$ to hold whenever there is a pair of sibling leaves $\langle a \,|\, p\rangle, \langle b \,|\, p\rangle \in T$ such that $a \neq b$, and $T' = \text{Join\_classes}(a, b, T)$. For all trees T, T' for atomic theories and every $n \in \mathbb{N}$, let $T \rightarrowtail^n T'$ hold if there is a sequence $T_1, \ldots, T_{n+1}$ of trees for atomic theories of the length $n+1$ such that $T_i \rightarrowtail T_{i+1}$ for all $1 \leq i \leq n$. We call such sequences *tree derivations*. Note that in particular $T \rightarrowtail^0 T'$ iff $T' = T$.

Let $\rightarrowtail^*$ be the reflexive and transitive closure of $\rightarrowtail$. Obviously $\rightarrowtail^* = \bigcup_{n\in\mathbb{N}} \rightarrowtail^n$, and whenever $T \rightarrowtail^* T'$ for any trees for atomic theories T and T', there exists a tree derivation starting with T and ending with T'. For a pair of terms $s, t \in D$ and an atomic theory Γ, we say that *there is a tree derivation of $s = t$ from Γ* if $T(\Gamma) \rightarrowtail^* T$ for some tree T in which there is such a leaf $\langle e \,|\, p\rangle$ that $s, t \in e$.

We now prove soundness and completeness of tree derivations w.r.t. the usual first-order semantics (Thm. 3.5.10) with the help of Lemma 3.5.9, which establishes soundness and completeness of tree derivations relatively to the C proof system.

3.5.9 Lemma.

(i) For every C derivation $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$, we have $T(\Gamma_1) \rightarrowtail^* T(\Gamma_n)$.

(ii) Let Γ be an atomic theory, let Δ and Π be sets of equalities such that $\Gamma \vDash_{\mathrm{e}} \Pi$, $\Pi \vDash_{\mathrm{e}} \Delta$, and $\Delta \cap \Gamma = \emptyset$. If there is a tree derivation of $s = t$ from $\Delta \cup \Gamma$, then $\Pi \cup \Gamma \vdash_{\mathrm{C}} s = t$.

Proof. (i) Take any C derivation $\Gamma_1, \ldots, \Gamma_n$. We prove by induction on i, that if $1 \leq i \leq n$, then $T(\Gamma_1) \rightarrowtail^* T(\Gamma_i)$, from which the proposition follows. The base case is immediate. For the induction step, assume that $T(\Gamma_1) \rightarrowtail^* T(\Gamma_i)$ for some $1 \leq i < n$. If the rule used to obtain Γ_{i+1} from Γ_i is 3.4.2(1), then $T(\Gamma_1) \rightarrowtail^* T(\Gamma_i) = T(\Gamma_{i+1})$ by Lemma 3.5.4(iii). Otherwise the rule applied to Γ_i is 3.4.2(2), thus there is $(\Rightarrow u = v) \in \Gamma_i$ and $\Gamma_{i+1} = (u = v), \Gamma_i$. Whence $T(\Gamma_{i+1}) = \text{Join\_classes}([u], [v], T(\Gamma_i))$ by Lemma 3.5.6 (equivalence classes are for the induced equivalence $\simeq_{\Gamma_i^{\mathrm{e}}}$). If $[u] \neq [v]$, there is a pair of sibling leaves with distinct labels in $T(\Gamma_1)$ by Lemma 3.5.4(iv), consequently $T(\Gamma_i) \rightarrowtail T(\Gamma_{i+1})$, thus $T(\Gamma_1) \rightarrowtail^* T(\Gamma_{i+1})$. If $[u] = [v]$, then $T(\Gamma_1) \rightarrowtail^* T(\Gamma_i) = T(\Gamma_{i+1})$.

(ii) The claim is clearly a consequence of the following proposition, which we prove by induction on k:

Let Γ be an atomic theory, let Δ and Π be sets of equalities such that $\Gamma \vDash_{\mathrm{e}} \Pi$, $\Pi \vDash_{\mathrm{e}} \Delta$, and $\Delta \cap \Gamma = \emptyset$. Let T be a tree such that $T(\Delta \cup \Gamma) \rightarrowtail^k T$, and assume there is a leaf $\langle e \,|\, p\rangle \in T$ with $s, t \in e$. Then $\Pi \cup \Gamma \vdash_{\mathrm{C}} s = t$.

For $k = 0$, take any Γ, Δ, Π satisfying the assumption, and note that we have a leaf $\langle e \,|\, r\rangle \in T(\Delta \cup \Gamma)$ with $s, t \in e$, i.e., $\Delta \cup \Gamma \vDash_{\mathrm{e}} s = t$ by Lemma 3.5.4(i). Every

equivalence satisfying $\Pi \cup \Gamma$ satisfies $\Delta \cup \Gamma$, because of the assumption $\Pi \models_{\mathrm{e}} \Delta$. Therefore $\Pi \cup \Gamma \models_{\mathrm{e}} s = t$, thus the single-element sequence containing only $\Pi \cup \Gamma$ is a C derivation of $s = t$ from $\Pi \cup \Gamma$.

Assume the proposition holds for $k \geq 0$. Take any Γ, Δ, Π, and T satisfying the assumption for $k+1$. Since $T(\Delta \cup \Gamma) \rightarrowtail^{k+1} T$, there is a pair of sibling leaves $\langle a \,|\, p\rangle, \langle b \,|\, p\rangle \in T(\Delta \cup \Gamma)$ such that $a \neq b$, and $\mathrm{Join\_classes}(a, b, T(\Delta \cup \Gamma)) \rightarrowtail^{k} T$.

By Lemma 3.4.8(i), there is a strictly monotonic C derivation $\Sigma_1, \ldots, \Sigma_n$ constructed only by the rule 3.4.2(1) so that $\Sigma_1 = \Delta \cup \Gamma$, and Σ_n is the 3.4.2(1)-saturated set $\Delta \cup \Gamma'$, where $\Gamma' = \mathrm{C}(\Delta \cup \Gamma) \cup \Gamma$. We have $T(\Sigma_n) = T(\Delta \cup \Gamma)$ from Lemma 3.5.4(iii). By Lemma 3.4.8(ii), there is a C derivation $S := \Sigma'_1, \ldots, \Sigma'_n$ such that $\Sigma'_i = (\Sigma_i \setminus \Delta) \cup \Pi$ for every $i = 1, \ldots, n$. In particular, $\Sigma'_1 = \Pi \cup \Gamma$, and $\Sigma'_n = \Pi \cup \Gamma'$.

By Lemma 3.5.4(v), there are terms $u, v, f(\vec{u}), f(\vec{v}) \in D$ such that $(\Rightarrow u = f(\vec{u}))$, $(\Rightarrow v = f(\vec{v})), (\Rightarrow f(\vec{u}) = f(\vec{v})) \in \Gamma'$ (we have omitted Δ as it contains no atomic clauses), $a = [u]$, and $b = [v]$. Thus by Lemma 3.5.6 and Lemma 3.5.4(iii), $\mathrm{Join\_classes}(a, b, T(\Delta \cup \Gamma)) = T(\Delta' \cup \Gamma) = T(\Delta' \cup \Gamma')$ for $\Delta' = \{u = v\} \cup \Delta$. Moreover, we can form a new C derivation S' by appending three more steps to S applying the rule 3.4.2(2) to the three clauses. The last element of S' is $\Pi' \cup \Gamma'$, where $\Pi' = \{u = f(\vec{u}), v = f(\vec{v}), f(\vec{u}) = f(\vec{v})\} \cup \Pi$.

Observe now that $\Pi' \models_{\mathrm{e}} \Delta'$, and recall that $T(\Delta' \cup \Gamma') \rightarrowtail^{k} T$. It is easy to verify that $\Gamma' \models_{\mathrm{e}} \Pi'$, $\Pi' \models_{\mathrm{e}} \Delta'$, and $\Delta' \cap \Gamma' = \emptyset$. We can now instantiate the induction hypothesis with Γ' in place of Γ, Δ' in place of Δ, and Π' in place of Π. Hence we obtain a C derivation of $s = t$ from $\Pi' \cup \Gamma'$, which we append to S' creating a C derivation of $s = t$ from $\Pi \cup \Gamma$. □

3.5.10 Theorem (Soundness and completeness of tree derivations). *Let Γ be an atomic theory, and let s, t be arbitrary terms of D. There exists a tree derivation of $s = t$ from Γ iff $\Gamma \models s = t$.*

Proof. By Thm. 3.4.6, it is sufficient to prove that there exists a tree derivation of $s = t$ from Γ iff $\Gamma \vdash_{\mathrm{C}} s = t$. The implication ($\rightarrow$) is a direct consequence of Lemma 3.5.9(ii) for $\Delta = \Pi = \emptyset$. For the implication ($\leftarrow$), assume there is a C derivation $\Gamma_1, \ldots, \Gamma_n$ such that $\Gamma_1 = \Gamma$, and $\Gamma_n \models_{\mathrm{e}} s = t$. Hence by Lemma 3.5.9(i), there is a tree derivation $T_1, \ldots, T_k$ for Γ such that $T_k = T(\Gamma_n)$. Thus by Lemma 3.5.4(i), there is a leaf $\langle e \,|\, r\rangle \in T_k$ with $s, t \in e$, i.e., $T_1, \ldots, T_k$ is a tree derivation of $s = t$ from Γ. □

3.5.11 Algorithmic properties of $\rightarrowtail^*$. While soundness is important from the algorithmic point of view, completeness is too coarse a property. It only assures that there is a way to iteratively apply Join_classes to a tree for an atomic theory to derive an equality. Actually, any tree derivation starting from a tree $T(\Gamma)$ can be extended so that it leads to a tree, where for every $s, t \in D$ there is a leaf whose label contains both s and t iff $\Gamma \models s = t$. This important property of tree derivations

is expressed rigorously in Thm. 3.5.12. In the theorem and its proof, we denote for every Γ the set $\{\, s = t \mid s, t \in D \text{ and } \Gamma \vDash_{\mathrm{c}} s = t \,\}$ by $\mathrm{C}^*(\Gamma)$.

3.5.12 Theorem. *Let Γ be an atomic theory, and let T be a tree for an atomic theory such that $T(\Gamma) \rightarrowtail^* T$. Then $T \rightarrowtail^* T(\mathrm{C}^*(\Gamma))$.*

Proof. The lemma is an immediate consequence of the following two auxiliary propositions:

(a) Let Γ be an atomic theory, and T be a tree for an atomic theory such that $T(\Gamma) \rightarrowtail^* T$. Then there is $\Delta \subseteq \mathrm{C}^*(\Gamma)$ such that $T = T(\Delta \cup \Gamma)$.

(b) Let Γ be an atomic theory, and let $\Delta \subseteq \mathrm{C}^*(\Gamma)$. Then $T(\Delta \cup \Gamma) \rightarrowtail^* T(\mathrm{C}^*(\Gamma))$.

Notice that since every $\Delta \subseteq \mathrm{C}^*(\Gamma)$ contains only equalitites, $(\Delta \cup \Gamma)^{\mathrm{e}} = \Delta \cup \Gamma^{\mathrm{e}}$. We will always use the latter form. Also note that for the same reason $T(\mathrm{C}^*(\Gamma)) = T(\simeq_{\mathrm{C}^*(\Gamma)^{\mathrm{e}}})$.

We prove (a) by induction on k for all T such that $T(\Gamma) \rightarrowtail^k T$. The basis is immediate with $\Delta = \emptyset$. For the inductive step, take any T such that $T(\Gamma) \rightarrowtail^{k+1} T$. Then there is T' such that $T(\Gamma) \rightarrowtail^k T'$, and $T' \rightarrowtail T$. Thus there is a pair of sibling leaves $\langle a \,|\, p\rangle, \langle b \,|\, p\rangle \in T'$ such that $a \neq b$, and $T = \mathrm{Join\_classes}(a, b, T')$. Further by the induction hypothesis, there is $\Delta \subseteq \mathrm{C}^*(\Gamma)$ such that $T' = T(\Delta \cup \Gamma)$. Therefore by Lemma 3.5.4(v), there are $(\Rightarrow s = f(\vec{s})), (\Rightarrow t = f(\vec{t})), (\Rightarrow f(\vec{s}) = f(\vec{t})) \in \Delta \cup \Gamma \cup \mathrm{C}(\Delta \cup \Gamma)$, such that $\Delta \cup \Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{s} = \vec{t}$, $a = [s]$, and $b = [t]$ (the equivalence classes are for $\simeq_{\Delta \cup \Gamma^{\mathrm{e}}}$). Consequently $T = T(\{s = t\} \cup \Delta \cup \Gamma)$ by Lemma 3.5.6. It is easy to check that $\Gamma \vDash_{\mathrm{c}} s = t$, hence we conclude $\{s = t\} \cup \Delta \subseteq \mathrm{C}^*(\Gamma)$.

We prove (b) by induction on n for all Δ such that $|D/{\simeq_{\Delta \cup \Gamma^{\mathrm{e}}}}| = n$. The base case is trivial as $|D/{\simeq_{\Delta \cup \Gamma^{\mathrm{e}}}}| > 0$. Take any n and Δ such that $|D/{\simeq_{\Delta \cup \Gamma^{\mathrm{e}}}}| = n + 1$. Assume first that there is a pair of sibling leaves $\langle a \,|\, p\rangle, \langle b \,|\, p\rangle \in T(\Delta \cup \Gamma)$ such that $a \neq b$. Consequently by Lemma 3.5.4(v), there are $(\Rightarrow s = f(\vec{s})), (\Rightarrow t = f(\vec{t}))$, $(\Rightarrow f(\vec{s}) = f(\vec{t})) \in \Delta \cup \Gamma \cup \mathrm{C}(\Delta \cup \Gamma)$, such that $\Delta \cup \Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} \vec{s} = \vec{t}$, $a = [s]$, and $b = [t]$ (the equivalence classes are for $\simeq_{\Delta \cup \Gamma^{\mathrm{e}}}$). Denote by T' the tree $T(\{s = t\} \cup \Delta \cup \Gamma) = \mathrm{Join\_classes}(a, b, T(\Delta \cup \Gamma))$ (the equality is by Lemma 3.5.6). One can verify that $|D/{\simeq_{\{s=t\} \cup \Delta \cup \Gamma^{\mathrm{e}}}}| = n$ using Lemma 3.3.11, and that $(s = t) \in \mathrm{C}^*(\Gamma)$. Therefore the induction hypothesis yields $T' \rightarrowtail^* T(\mathrm{C}^*(\Gamma))$. Since $T(\Delta \cup \Gamma) \rightarrowtail T'$, we have $T(\Delta \cup \Gamma) \rightarrowtail^* T(\mathrm{C}^*(\Gamma))$.

The remaining case is that (b) holds when there are no pairs of sibling leaves with distinct labels in $T(\Delta \cup \Gamma)$. Denote the atomic theory $\Delta \cup \Gamma \cup \mathrm{C}(\Delta \cup \Gamma)$ by Γ', and observe that by Lemma 3.5.4(iii), $T(\Delta \cup \Gamma) = T(\Gamma')$. Also notice that by Lemma 3.5.4(iv) for all $(\Rightarrow s = t) \in \Gamma'$, we have $\Delta \cup \Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = t$, thus $\Gamma'^{\mathrm{e}} \vDash_{\mathrm{e}} s = t$ since $\Gamma'^{\mathrm{e}} = \Delta \cup \Gamma^{\mathrm{e}}$ as there are no equalities in $\mathrm{C}(\Delta \cup \Gamma)$. Hence for every $(\Rightarrow s = f(\vec{t})) \in \Gamma'$, $\langle [s], f, [t_1], \ldots, [t_n], \emptyset\rangle = \langle [f(\vec{t})], f, [t_1], \ldots, [t_n], \emptyset\rangle \in T(\simeq_{\Gamma'^{\mathrm{e}}})$, thus $T(\Gamma') = T(\simeq_{\Gamma'^{\mathrm{e}}})$. Also, the atomic theory Γ' possesses both properties of weak C saturation, whence $\Delta \cup \Gamma^{\mathrm{e}} \vDash_{\mathrm{e}} s = t$ iff $\Gamma' \vDash_{\mathrm{c}} s = t$ for all $s, t \in D$ by Lemma 3.4.5. Thus $T(\Delta \cup \Gamma) = T(\simeq_{\Delta \cup \Gamma^{\mathrm{e}}}) = T(\simeq_{\mathrm{C}^*(\Gamma)^{\mathrm{e}}}) = T(\mathrm{C}^*(\Gamma))$. □

3.5.13 Corollaries of Theorem 3.5.12. One can easily verify that by Lemma 3.5.4(i) and Thm. 3.3.8(ii), for every $s, t \in D$ there is a leaf in $T(\mathrm{C}^*(\Gamma))$ whose label contains both s and t iff $\Gamma \models s = t$. Inspection of the proof reveals that this tree can also be built from any (weakly) C-saturated theory derivable from Γ. The theorem implies that for any theory Γ, we can reach $T(\mathrm{C}^*(\Gamma))$ starting from $T(\Gamma)$, and that we can do so regardless of which pair of distinct labels of sibling leaves we decide to join in any particular step. (We could perhaps say that the relation $\rightarrowtail$ is a confluent "tree-rewriting" system.) Further, $T(\mathrm{C}^*(\Gamma))$ is the only tree with no sibling leaves reachable from $T(\Gamma)$, since for every such tree T there is no T' for which $T \rightarrowtail T'$, therefore $T \rightarrowtail^* T(\mathrm{C}^*(\Gamma))$ holds only if $T = T(\mathrm{C}^*(\Gamma))$.

Note also that since in every step of a tree derivation the number of equivalence classes decreases (as noted in the proof), we reach the tree $T(\mathrm{C}^*(\Gamma))$ after less than $|D|$ steps. However, the complexity of the imperative algorithm from the following section is higher that linear in $|D|$, since it involves maintenance of a tree representing data structure representing.

3.6 An Efficient Imperative Algorithm for Congruence Closure

As noted in the last paragraph of the previous section, we need at most $|D|$ applications of the function Join_classes to set theoretical trees, to obtain the tree $T(\mathrm{C}^*(\Gamma))$, which enables us to decide logical consequence of every equality among the terms of D from Γ. In an actual imperative implementation, the application of Join_classes is not an $O(1)$ operation, and involves traversal of some part of the tree. If implemented naïvely, the time complexity of the algorithm would be $O(|D|^2)$. The most efficient algorithms known for congruence closure need time $O(n \log n)$ on average. Figures 3.4, 3.5, and 3.7 outline in abstract terms our $O(n \log n)$ implementation of the congruence closure algorithm.

3.6.1 Data structures. The data structures of our imperative algorithm in Fig. 3.4 are designed to represent a tree for an atomic theory. The algorithm (itself in Fig. 3.7) operates by side-effects on the global variable *tree* which is implemented as an array of ten-tuples indexed by values of type Node. The values of type Node correspond to the nodes in the tree $Tr(\Gamma)$, but they are implemented as numbers so they can be used as indices to the array. The structure of the tree is implemented as usual: The root of the tree is at the index 0, and for a node n, *tree*$[n]$ contains the index of the parent and the index of the first child, which is strung together with its siblings through the fields *next-sibling* and *prev-sibling*. We need a doubly-linked list of siblings so that the removal of a child from *children*$[p]$ can be implemented in time $O(1)$.

The nodes of the tree $Tr(\Gamma)$ are labeled by $\emptyset$ (root only), function symbols (parents of leaves), or equivalence classes of $\simeq_{\Gamma^{\mathrm{e}}}$ (all other nodes). In the implementation (the type Label), equivalence classes are replaced by values of type Representative, which is a subtype of Node. When the tree is initally built for the domain D, the

type Node ▷ $Tr(\Gamma)$
Representative $\subseteq$ Node ▷ the set of representatives of classes in $D/{\simeq_{\Gamma^e}}$
Label = Representative $\cup$ Func-Sym $\cup \{\emptyset\}$ ▷ labels of nodes
prop *parent*: Node $\to$ Node ▷ $parent[\langle \ell \,|\, p\rangle] = p$
label: Node $\to$ Label ▷ $label[\langle \ell \,|\, p\rangle] = \ell$
children: Node $\to \mathcal{P}$(Node) ▷ $children[p] = \{\, \langle \ell \,|\, p\rangle \mid \langle \ell \,|\, p\rangle \in Tr(\Gamma) \,\}$
uses: Label $\to \mathcal{P}$(Node) ▷ $uses[\ell] = \{\, \langle \ell \,|\, p\rangle \mid \langle \ell \,|\, p\rangle \in Tr(\Gamma) \,\}$
cwl: $\mathcal{P}$(Label $\times$ Node $\times$ Node) ▷ $cwl = \{\, \langle \ell, p, \langle \ell \,|\, p\rangle\rangle \mid \langle \ell \,|\, p\rangle \in Tr(\Gamma) \,\}$
var *tree*: **array** [Node] **of record** *parent*, *first-child*: Node; *num-children*: $\mathbb{N}$; *label*: Label;
next-sibling, *prev-sibling*, *next-use*, *prev-use*, *next-conflict*, *prev-conflict*: Node **end**
first-use: **array** [Representative] **of** Node
cwl-implem: **hash** [Representative, Node] **of** Node
with conflict lists in *tree.next-conflict*, *tree.prev-conflict*

Figure 3.4. Efficient implementation of the congruence closure algorithm (data structures and properties).

equivalence class of each term $f(t_1, \ldots, t_n) \in D$ contains only that term. The class is identified with the node $\langle f, \ell_1, \ldots, \ell_n, \emptyset\rangle$ where ℓ_1, …, ℓ_n are of type Representative (and consequently of type Label), and are identified in that order with the classes $[t_1]$, …, $[t_n]$. This correspondence of equivalence classes and representatives is not maintained during the run of the algorithm. Details will be explained later.

The algorithm accesses the global variable *tree* (along with auxiliary global variables *first-use* and *cwl*) through a set of (imperative) accessing/modifying properties (similar to object properties in Delphi or C#, but more general), which are presented on an abstract level in Fig. 3.4. Most of them behave like finite functions, i.e., arrays. Applications $parent[n]$ and $label[n]$ are merely abbreviations of $tree[n].parent$ and $tree[n].label$ respectively. The *children* property is a convenient abstraction of linked lists of children described above, so that the application $children[p]$ yields the finite set $\{\, \langle e \,|\, p\rangle \mid \langle e \,|\, p\rangle \in Tr(\simeq) \,\}$ of children of the node p. The property *uses* is similar: $uses[\ell]$ yields the finite set $\{\, \langle \ell \,|\, p\rangle \mid \langle \ell \,|\, p\rangle \in Tr(\simeq) \,\}$ of nodes with the label ℓ. The corresponding doubly-linked list starts with the node $first\text{-}use[\ell]$, and its member nodes are connected through the fields *next-use*, *prev-use*.

The property *cwl* (*c*hild *w*ith *l*abel) behaves like an (imperative) relation which can be viewed as a partial map yielding for a label ℓ and a node p the child q of p with the label a if it exists, i.e., if $\langle \ell, p, q\rangle \in cwl$. The last corresponds to $q = \langle \ell \,|\, p\rangle \in Tr(\simeq)$. The partial map *cwl* is rather sparse. For a tree with n nodes, the map is defined on $O(n)$ pairs $\langle \ell \,|\, p\rangle$ out of possible $O(n^2)$. In order to obtain the average complexity $O(n \log n)$ of the congruence closure algorithm, it is necessary that the child q is yielded on average in time $O(1)$. Note that the naïve traversal of the set $children[p]$ gives the access time $O(n)$. Probably the best implementation of such a map is by a hash table of the size $O(n)$ (the variable *cwl-implem* in Fig. 3.4)

```
procedure ADD-ATOMIC-CLAUSE(s, t)
  m ← REPRESENT(s)
  n ← FIND-OR-ADD-TERM(t)
  _, _ ← FIND-OR-ADD-CHILD(n, m)

function REPRESENT(t)
  n ← FIND-OR-ADD-TERM(t)
  return label[tree[n].first-child]

function FIND-OR-ADD-TERM(f(s⃗))
  p ← 0
  for i ← arity[f], ..., 1
    n ← REPRESENT(s_i)
    _, p ← FIND-OR-ADD-CHILD(p, n)
  is-new, n ← FIND-OR-ADD-CHILD(p, f)
  if is-new
    _, _ ← FIND-OR-ADD-CHILD(n, n)
  return n

function FIND-OR-ADD-CHILD(p, a)
  if exists n: ⟨a, p, n⟩ ∈ cwl
    return ⟨false, n⟩
  else
    take a new node n
    label[n] ← a
    ADD(p, n)
    return ⟨true, n⟩

procedure ADD(p, n)
  parent[n] ← p
  add n to children[p]
  add n to uses[label[n]]
  add ⟨label[n], p, n⟩ to cwl

procedure REMOVE(n)
  if n ≠ 0
    remove ⟨label[n], parent[n], n⟩ from cwl
    remove n from uses[label[n]]
    remove n from children[parent[n]]
```

Figure 3.5. Efficient implementation of the congruence closure algorithm (initialization and auxiliary procedures).

with hash conflicts doubly-linked through the nodes in the array *tree*. However, at a negligible loss of efficiency we can also choose to implement *cwl* by a balanced tree or by a trie.

3.6.2 Initialization of data structures. The procedures in the left column of Fig. 3.7 are used to initialize the data structures. First, we use the function FIND-OR-ADD-TERM on all terms of the domain D to build the initial tree. We then use the procedure ADD-ATOMIC-CLAUSE on all atomic clauses, as well as all equalities in the D-bounded atomic set Γ to which we wish to apply the congruence closure. Note that the set Γ and its counterpart with all equalities changed to atomic clauses are equivalent. We will now describe the initialization functions in slightly more detail.

For a term $f(\vec{s}) \in D$, FIND-OR-ADD-TERM is called recursively (indirectly via REPRESENT) on each argument s_i of the top-most function symbol f in the order last to first to ensure that the arguments (if there are any) are already in the tree, and to find out their representatives n_i. A path is constructed or partially looked up in the tree (by calls to FIND-OR-ADD-CHILD) from the node 0 so that the nodes in the path are labeled by the representatives of arguments. (Note that a part of the path may be already present in the tree, for instance, if we are adding the term $g(a, b)$, and the term $f(a, b)$ was added before, the path $\langle$representative of a, representative of $b, \emptyset\rangle$ is already in the tree, FIND-OR-ADD-CHILD finds the existing nodes, and

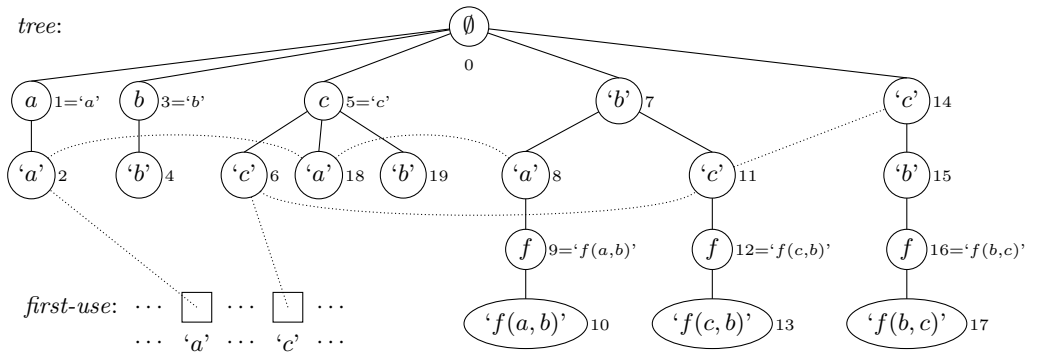


Figure 3.6. An example tree maintained by the congruence closure algorithm.

we reuse them.)

Once we have a path labeled by the representatives of the arguments $\vec{s}$, we use Find-or-Add-Child on the last node of the path and the function symbol f. During the initialization, this creates a new node (*is-new* = **true**), and its index n serves as the representative of the single-element equivalence class containing $f(\vec{s})$. We then add a child to this node labeled by n. This is analogous to $[f(\vec{s})]$ being the final element of the path $\langle [f(\vec{s})], f, [s_1], \ldots, [s_n], \emptyset \rangle$ in $Tr(\simeq_{\Gamma^e})$. However, the function Find-or-Add-Term can be used not only for creating paths representing terms, but also for finding representatives (through a call to Represent). In that case, the node labeled by f and its child already exists in the tree (*is-new* = **false**). We then do not add any children, only return the index of the node labeled by f.

The procedure Add-Atomic-Clause creates a path in the tree analogous to $\langle [s], f, [t_1], \ldots, [t_n], \emptyset \rangle$ in $T(\Gamma)$ by adding a child labeled with the representative of s to the node representing t.

Actual addition of a node into the structure of the tree and into *uses* of its label is performed by the procedure Add, which is defined abstractly in Fig. 3.5 to hide unnecessary details of updating linked lists. The same holds for the procedure Remove, which is employed in the main algorithm to remove a node from the tree and *uses*.

3.6.3 Example (Initialized data structures). Fig. 3.6 depicts a graphical representation of the data structure for the theory Γ_1 in the domain D as used defined in Par. 3.5.2 and Par. 3.5.3. The tree is very similar to the tree $T(\Gamma_1)$ from Fig. 3.2(a). Number to the left of a node indicates the index of the node. Indices of nodes labeled by function symbols are used as representatives of terms. We use the notation 't' to denote the representative of a term t. We prefer the notation over direct usage of representative numbers for obvious reasons. Dotted lines in the figure depict the *uses* linked lists. Only the lists for the representatives 'a' = 2 and 'c' = 6 are shown.

The tree was be built by calling Find-or-Add-Term on terms of D in the order

```
procedure Congruence-Closure
  while there are sibling leaves with labels a, b
    Join-Classes(a, b)

procedure Join-Classes(a, b)
  assume w.l.o.g. that |uses[a]| ≥ |uses[b]|
  class[b] ← a
  for each n in uses[b]
    if label[n] = b
      p ← parent[n]
      Remove(n)
      if exists m: ⟨a, p, m⟩ ∈ cwl
        Remove(m)
        n ← Merge-Nodes(m, n)
      label[n] ← a
      Add(p, n)

function Merge-Nodes(p, q)
  assume w.l.o.g. that |children[p]| ≥ |children[q]|
  for each n in children[q]
    c ← label[n]
    Remove(n)
    if exists m: ⟨c, p, m⟩ ∈ cwl
      Remove(m)
      n ← Merge-Nodes(m, n)
    Add(p, n)
  return p
```

Figure 3.7. Efficient implementation of the congruence closure algorithm (the main algorithm).

a, b, c, $f(a,b)$, $f(c,b)$, $f(b,c)$. Consider, for instance, the term $f(c,b)$. The representative of b was first looked up to be 'b' = 3. A child of 0 labeled with 'b' was already present (the node 7). A new node (11) was created as its child, and labeled with 'c'. Another new node (12) was created as the child of 11, and labeled with f. Since the node was new, the leaf node 13 with the label 12 = '$f(c,b)$' was added as the first child of 12.

3.6.4 The working of the algorithm. The main procedure Congruence-Closure of the algorithm in Fig. 3.7 looks for two sibling leaves with labels some a and b, and joins them, in a way analogous to the function Join_classes from Par. 3.5.5, by calling the procedure Join-Classes(a,b). The only difference is that the former works with equivalence classes, while the latter with their representatives.

Join-Classes finds all occurrences of the label b in the tree by traversing the set *uses*$[b]$ of all nodes with the label b. For every node $\langle b \,|\, p\rangle$ in the set, its descendants $q +\!\!+ \langle b \,|\, p\rangle$ are merged by calling Merge-Nodes$(\langle a \,|\, p\rangle, \langle b \,|\, p\rangle)$ with the descendants $r +\!\!+ \langle a \,|\, p\rangle$ of the sibling node $\langle a \,|\, p\rangle$ of $\langle b \,|\, p\rangle$ if there is one in the tree. The label b of $\langle b \,|\, p\rangle$ is renamed to a, because a will be the new representative of the union of classes represented by a and b respectively.

The procedure Merge-Nodes(p,q) traverses all children $\langle c \,|\, q\rangle$ of q making $\langle c \,|\, q\rangle$ a child of p unless there is a child $\langle c \,|\, p\rangle$ of p. In that case, the procedure first recursively merges $\langle c \,|\, q\rangle$ with $\langle c \,|\, p\rangle$, and then makes the result a child of p.

3.6.5 Example (Join-Classes). In Fig. 3.8, we see how Join-Classes('a','c') modifies the data structure from Fig. 3.6. The index 'a' was chosen as the common

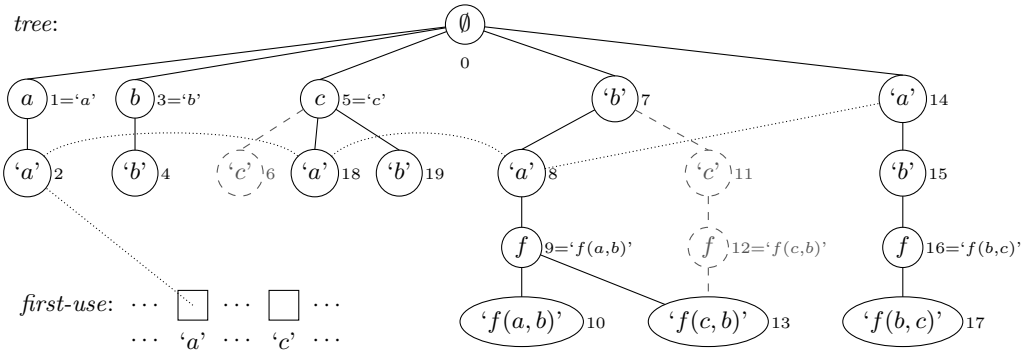


Figure 3.8. The tree from Fig. 3.6 after JOIN-CLASSES(‘a’, ‘c’) has been performed.

representative. All occurences of ‘c’ were replaced by ‘a’ except in nodes 6 and 11, which were removed from the tree. New occurences of ‘a’ were added to $uses[a]$.

3.6.6 Correctness and termination. Unlike in algorithms from [NO80, DST80], but similarly to [NO03], there is no union–find-set data structure to record the history of replacement of representatives in our version of the congruence closure algorithm. Therefore the correspondence of data structures and set-theoretical trees from the previous section is not obvious.

During the run of the algorithm, there is no canonical representative of a term. Take, for instance, the term c (more precisely $c()$) in our example in Fig. 3.8. The node $5 =$ ‘c’ has two children, none of which is labeled ‘c’ (the node 6 was removed). Not taking into account the history of the tree, we could consider ‘b’ (the label of the node 19) as well as ‘a’ (the label of the node 18) to be representatives of the term c. The function REPRESENT($c()$) returns the label of that node which happens to be the first member of the linked list of children of the node 5.

However, we can easily compensate for the lack of direct correspondence between equivalence classes and representatives by Lemma 3.6.7. The lemma relates the imperative algorithm to the set-theoretical one in such a way that Thm. 3.5.10, Thm. 3.5.12, and Par. 3.5.13 immediatelly apply also to the imperative algorithm. The fact that the lemma is formulated only for atomic theories with no equalities is not significant, since the atomic theory $\Gamma \cup \{\, (\Rightarrow s = t) \mid (s = t) \in \Gamma^{\mathrm{e}} \,\}$ is a conservative extenstion of Γ. This proves correctness and termination of the algorithm.

Note that once the algorithm has terminated, there are no sibling leaves, thus there is only one representative for each term. In the set-theoretical case, we have $\Gamma \vDash s = t$ iff there is a leaf $\langle e \,|\, p \rangle$ in a tree T such that $T(\Gamma) \rightarrowtail^* T$, and $s, t \in e$. In the imperative case, $\Gamma \vDash s = t$ iff REPRESENT(s) = REPRESENT(t) once the data structure was brought to a state with no sibling leaves, which is possible by Thm. 3.5.12.

3.6.7 Lemma (Invariant of the imperative algorithm). *Let Γ be an atomic theory containing no equalities. Before every iteration of the* **while** *loop in the procedure* Congruence-Closure, *there is an atomic theory Γ' such that $T(\Gamma) \rightarrowtail^* T(\Gamma')$, and there is a bijection r between the set of representatives used as labels in the data structure and the set $D/{\simeq_{\Gamma'^{\mathrm{e}}}}$. Under this bijection, the tree represented by the current state of the data structure is identical to $Tr(\Gamma')$.*

Proof sketch. The lemma is proved by induction on the number of iterations of the **while**-loop. To prove the basis, recall that after adding all terms of D into the data structure, the node returned by Find-or-Add-Term(t) is the representative of the term t (Par. 3.6.2). Since there are no equalities in Γ, $\simeq_{\Gamma^{\mathrm{e}}}$ is an identity relation, hence $[t] = \{t\}$. We define the bijection r so that $r(\text{Find-or-Add-Term}(t)) = \{t\}$, and easily show the data structure and the tree $Tr(\Gamma)$ identical under r.

In the induction step, we take the bijection r' and atomic theory Γ'' for the state of the data structure before the last call to Join-Classes for some labels a and b of a pair of sibling leaves. We define $r(c) = r'(c)$ if $a \neq c \neq b$, and $r(a) = r(b) = r'(a) \cup r'(b)$, and take $\Gamma' = (s = t), \Gamma''$ for some terms $s \in r'(a)$, $t \in r'(b)$. By Lemma 3.5.6, $T(\Gamma') = \text{Join\_classes}(r'(a), r'(b), T(\Gamma''))$. The proposition now holds provided the procedure Join-Classes behaves analogously to the function Join_classes, which should be clear from Par. 3.6.4. □

3.6.8 Complexity of the algorithm. In both Join-Classes and Merge-Nodes, it is crucial that the **for each** loops traverse the smaller set, and access the children from the larger set in the constant time. This assures that each out of the n nodes in the tree is removed from some smaller set and added to a larger set at most $O(\log n)$ times (the size of the set obtained by the joined and merged nodes is at least double the size of the smaller set). Thus the time complexity of the algorithm is on $O(n \log n)$ (on average because of the hashing).

Suppose that the congruence closure tree is initialized to decide logical consequence from an atomic theory Γ in a domain D. The domain D is partially ordered by the subterm relation. There are possibly several maximal terms which are not subterms of any other member of D. The number of nodes n in the congruence closure tree is linear in the sum of the cardinality of Γ and the total length of maximal terms in the domain D. If D is the domain of Γ, then n is linear in the total length of sentences in Γ.

3.7 Related Work

Nelson and Oppen in [NO80] prove correctness of an algorithm which uses congruence closure to decide equalitites. However, the algorithm is formulated in a slightly different manner: Terms are represented by a directed acyclic graph where nodes represent terms. Each node is labeled with a function symbol. Edges from a node lead to all its direct subterms. The equivalence relation on terms is not represented in the graph. A less efficient, $O(n^2)$, version of the congruence closure algorithm

is discussed primarily in [NO80], although with frequent references to the efficient algorithm of [DST80]. On the other hand, [DST80] refers to [NO80] for the proof of correctness. Our discussion is much more detailed, and features a specialized formal system. We prove the correctness of our version of the algorithm relatively to this formal system.

The representation of terms used in our version of the algorithm differs from those in both [NO80] and [DST80]. Our version was inspired by [NO03]. The representation of terms used in [NO03] is based on currying, while ours is closer to the reduction of n-ary functions to unary using a pairing function, as used in the theory of computability. Our representation is in fact similar to the one obtained in [DST80] by reduction of graphs to outdegree 2. We however (similarly to [NO03]) obtain our representation directly.

We consider our representation more straightforward than those of [DST80, NO03] and facilitating understanding of the algorithm. We view the domain of terms as a tree. Branches of the tree merge, as the equivalence relation on terms is extended with new equalities. When two branches fully merge, a new equality is derived from congruence. This simple process is obscured in other representations.

4 Congruence Closure and Closed Clauses

We have described the congruence closure algorithm in detail in the previous section. We will now use the algorithm as a core of a decision procedure for the validity of sequents $\Gamma \Rightarrow \Delta$ where Γ and Δ are sets of arbitrary quantifier-free sentences. We will reduce this problem to logical consequence $\Pi \vDash s = t$ of an equality $s = t$ from a set of closed clauses Π. Recall the definition of closed clauses from Par. 3.2.1, and note that we will drop the adjectives "closed" and "quantifier-free" also in this chapter.

This decision problem is obviously NP-complete, and our decision procedure will be exponential. It is therefore not very suitable for use in a proof assistant, where we favor responsiveness over completeness. However, procedures deciding satisfiability of formulas in propositional logic with equality are of practical interest in areas such as verification of VLSI circuits.

The currently prevalent approach (see, e.g., [GHN04]) is driven by the propositional structure of formulas. We have attempted to absorb propositional structure to equality. We will show in Sect. 4.1, that this is fully possible for a special case where Π contains only Horn clauses, i.e., clauses with at most one equality in the consequent. Hence, logical consequence $\Pi \vDash s = t$ is decidable in time $O(n \log n)$. Feasible decidability makes this special case interesting in its own right.

General clauses can be reduced to equalities with the help of an axiom schema which is symmetric to congruence. We call this schema anticongruence, and we will define it in Sect. 4.3. We will then, in Sect. 4.4, convert any sequent $\Gamma \Rightarrow \Delta$ in a standard way to an equivalent set of general clauses, and subsequently to a set of equalities and anticongruence axioms. We will then extend the C proof system of Par. 3.4.2 to a CA proof system for sets of equalities under anticongruence. An algorithm implementing the CA proof system will be described in Sect. 4.6. Finally, Sect. 4.7 will conclude this chapter with remarks on relationship to other approaches to the decision of satisfiability of sets of general clauses with equality.

This chapter is an adaptation of the paper [Kľu05].

4.1 Horn Clauses

A *Horn clause* is a clause $\Gamma \Rightarrow \Delta$ with the consequent Δ containing at most one atomic formula. We will show in this section that the congruence closure algorithm is capable of deciding logical consequence $\Pi \vDash \vec{s} = \vec{t}$ for Π consisting of Horn clauses. This is possible through a suitable transformation of the problem into logical consequence of a single equality from an atomic theory.

We will start with two simple lemmas, which are more general than we need in this section: For any set of sentences Π, Lemma 4.1.1 reduces logical consequence $\Pi \models \Sigma$ to satisfiability of the set $\Pi, (\Sigma \Rightarrow)$. Lemma 4.1.2 enables transformation of any set of clauses Π (not necessarily Horn clauses) into a set of clauses $\Pi^{\mathtt{1}}$ in which consequents of all clauses are non-empty. Empty consequents are replaced with ones containing a single equality $\mathtt{0} = \mathtt{1}$ for a new constant $\mathtt{1}$. The original set Π is unsatisfiable iff $\mathtt{0} = \mathtt{1}$ is a logical consequence of $\Pi^{\mathtt{1}}$.

We will then be ready to reduce sets of Horn clauses to equivalent sets of equalities in Par. 4.1.3 and prove the equivalence of this reduction in Thm. 4.1.4. Finally, in Par. 4.1.5, we outline an algorithm deciding $\Pi \models \vec{s} = \vec{t}$ for sets of Horn clauses Π.

4.1.1 Lemma (Elimination of multiple consequences). *Let Π and Σ be sets of sentences in a functional language $\mathcal{L}$, and let Σ be finite. Then $\Pi \models_{\mathrm{c}} \Sigma$ iff $\Pi, (\Sigma \Rightarrow) \models_{\mathrm{c}} (\Rightarrow)$.*

Proof. Recall that by the definitions in Paragraphs 2.3.3 and 3.3.2, the empty clause $(\Rightarrow)$ is false in every model in both the standard and relational semantics. The q-consequence $\Pi \models_q (\Rightarrow)$ is therefore equivalent to q-unsatisfiability of the set Π (and analogically for logical consequence and unsatisfiability).

In the direction $(\rightarrow)$, assume $\Pi \models_{\mathrm{c}} \Sigma$. To prove c-unsatisfiability of $\Pi, (\Sigma \Rightarrow)$, take any congruence $\simeq$. If $\simeq$ satisfied $\Pi, (\Sigma \Rightarrow)$, then $\simeq$ would satisfy Π, but it would not satisfy Σ by the definition of $\simeq \models_{\mathrm{c}} (\Sigma \Rightarrow)$, which contradicts the assumption.

For the direction $(\leftarrow)$, assume $\Pi, (\Sigma \Rightarrow)$ is c-unsatisfiable, and take any congruence $\simeq$ satisfying Π. The congruence does not satisfy $\Pi, (\Sigma \Rightarrow)$, thus by the assumption, it does not satisfy $(\Sigma \Rightarrow)$, hence it satisfies Σ. □

4.1.2 Lemma (Elimination of empty consequents). *Let Π be a set of clauses in a functional language $\mathcal{L}$, let $\mathtt{1}$ be a new constant, and let $\Pi^{\mathtt{1}}$ denote the set*

$$\{\Gamma \Rightarrow \mathtt{0} = \mathtt{1} \mid (\Gamma \Rightarrow) \in \Pi\} \cup \{\Gamma \Rightarrow \Delta \mid (\Gamma \Rightarrow \Delta) \in \Pi \text{ and } \Delta \neq \emptyset\}.$$

Then $\Pi \models_{\mathrm{c}} (\Rightarrow)$ iff $\Pi^{\mathtt{1}} \models_{\mathrm{c}} \mathtt{0} = \mathtt{1}$.

Proof. In the direction $(\rightarrow)$, we assume $\Pi^{\mathtt{1}} \not\models_{\mathrm{c}} \mathtt{0} = \mathtt{1}$, and prove that Π is satisfiable. By the assumption, there is a c-model $\simeq^{\mathtt{1}}$ of $\Pi^{\mathtt{1}}$ such that $\simeq^{\mathtt{1}} \not\models \mathtt{0} = \mathtt{1}$. The congruence $\simeq^{\mathtt{1}}$ is over $D_{\mathcal{L}[\mathtt{1}]}$, and its restriction $\simeq$ to the domain $D_{\mathcal{L}}$ is a congruence over $D_{\mathcal{L}}$. We show $\simeq \models \Pi$ by taking any clause $\Gamma \Rightarrow \Delta$ from Π, and by considering two cases: If Δ is empty, then $\Gamma \Rightarrow \mathtt{0} = \mathtt{1}$ is in $\Pi^{\mathtt{1}}$, and thus $\simeq^{\mathtt{1}} \models \Gamma \Rightarrow \mathtt{0} = \mathtt{1}$. Since $\simeq^{\mathtt{1}} \not\models \mathtt{0} = \mathtt{1}$, we get $\simeq^{\mathtt{1}} \not\models \Gamma$, thus $\simeq \not\models \Gamma$, and hence $\simeq \models \Gamma \Rightarrow$. If Δ is non-empty, then the clause is in Π, and from $\simeq^{\mathtt{1}} \models \Gamma \Rightarrow \Delta$, we directly get $\simeq \models \Gamma \Rightarrow \Delta$.

In the direction $(\leftarrow)$, we assume that Π has a c-model $\simeq$, and prove $\Pi^{\mathtt{1}} \not\models_{\mathrm{c}} \mathtt{0} = \mathtt{1}$. The relation $\simeq$ is over $D_{\mathcal{L}}$, which is a domain in $\mathcal{L}[\mathtt{1}]$. By Lemma 3.3.5, $\simeq$ can be expanded to a congruence $\simeq^{\mathtt{1}}$ over $D_{\mathcal{L}[\mathtt{1}]}$ such that $\simeq^{\mathtt{1}} \not\models \mathtt{0} = \mathtt{1}$. It thus remains to show $\simeq^{\mathtt{1}} \models \Pi^{\mathtt{1}}$. Take any clause $\Gamma \Rightarrow \Delta$ in $\Pi^{\mathtt{1}}$. If the clause is also in Π, then $\simeq \models \Gamma \Rightarrow \Delta$, and hence $\simeq^{\mathtt{1}} \models \Gamma \Rightarrow \Delta$ by coincidence. Otherwise $\Delta = \{\mathtt{0} = \mathtt{1}\}$, and

$\Gamma \Rightarrow$ is in Π. Since $\simeq \vDash \Gamma \Rightarrow$, we have $\simeq \nvDash \Gamma$, by coincidence we get $\simeq^{\mathtt{1}} \nvDash \Gamma$, and hence $\simeq^{\mathtt{1}} \vDash \Gamma \Rightarrow \mathtt{0} = \mathtt{1}$. □

4.1.3 Reduction to atomic theories. According to the above lemmas, decision of logical consequence $\Pi \vDash \vec{s} = \vec{t}$ of equalities $\vec{s} = \vec{t}$ from a set of Horn clauses Π is reducible to decision of unsatisfiability of another set of Horn clauses $\Pi, (\vec{s} = \vec{t} \Rightarrow)$. That, in turn, is reducible to decision of logical consequence $(\Pi, (\vec{s} = \vec{t} \Rightarrow))^{\mathtt{1}} \vDash \mathtt{0} = \mathtt{1}$ of a single equality from a suitable set of Horn clauses, each with a non-empty consequent. We now show how to further reduce the problem to logical consequence $\Gamma \vDash \mathtt{0} = \mathtt{1}$ of an equality from an atomic theory Γ. This last problem is decidable by the congruence closure algorithm.

Let us fix a functional language $\mathcal{L}$. Take any finite set Π of closed Horn clauses with non-empty consequents. We extend $\mathcal{L}$ to a functional language $\mathcal{L}[\mathcal{C}]$, and form an atomic theory Γ by the following procedure: Initially $\mathcal{C}$ and Γ are empty. For each Horn clause $s_1 = t_1, \ldots, s_n = t_n \Rightarrow u = v$ of Π,

1. add a new n-ary function symbol g to $\mathcal{C}$;
2. add equalities $g(\vec{s}) = u$ and $g(\vec{t}) = v$ to Γ.

Intuitively, a congruence satisfying the atomic theory Γ satisfies not only $g(\vec{s}) = u$ and $g(\vec{t}) = v$, but also the congruence axiom $\vec{s} = \vec{t} \Rightarrow g(\vec{s}) = g(\vec{t})$, thus satisfying the original clause $\vec{s} = \vec{t} \Rightarrow u = v$. In the converse direction, we can extend any congruence satisfying $\vec{s} = \vec{t} \Rightarrow u = v$ to satisfy the congruence axiom and the two additional equalities. This is proved in the following theorem.

4.1.4 Theorem. *Let $\mathcal{L}$, $\mathcal{C}$, Π, and Γ be as in Par. 4.1.3. Then for terms $s, t \in D_{\mathcal{L}}$, $\Pi \vDash_{\mathrm{c}} s = t$ iff $\Gamma \vDash_{\mathrm{c}} s = t$.*

Proof. The theorem will immediately follow in we show that Γ is a c-conservative extension of Π for equalities among the terms of $\mathcal{L}$. To show that Γ is an extension of Π, take any congruence $\simeq$ satisfying Γ, and any Horn clause $(\vec{s} = \vec{t} \Rightarrow u = v) \in \Pi$. Assume $\simeq \vDash \vec{s} = \vec{t}$. Then $\simeq \vDash g(\vec{s}) = g(\vec{t})$ for g associated with the clause since $\simeq$ is a congruence. Consequently, we get $\simeq \vDash u = v$ from $\simeq \vDash \Gamma$.

For c-conservativity over equalities among the terms of $\mathcal{L}$, take any equality $s = t$ of $\mathcal{L}$ such that $\Gamma \vDash_{\mathrm{c}} s = t$, and any congruence $\simeq$ over the terms of $\mathcal{L}$ satisfying Π. We wish to prove that $\simeq$ satisfies $s = t$. Let us denote the set of terms $D_{\mathcal{L}} \cup \{g(\vec{s}) \mid (g(\vec{s}) = u) \in \Gamma\}$ by D. If $f(\vec{s}) \in D$, then either $f(\vec{s}) \in D_{\mathcal{L}}$, and then $\vec{s} \in D_{\mathcal{L}} \subseteq D$, or f is some $g \in \mathcal{C}$, and then $\vec{s}$ occurred in Π, thus again $\vec{s} \in D_{\mathcal{L}} \subseteq D$. Therefore, D is a domain in $\mathcal{L}[\mathcal{C}]$.

For every term $u \in D_{\mathcal{L}}$, let $\mathrm{G}(u)$ be the set of all $g(\vec{s})$ for which there is a term $v \simeq u$ such that $(g(\vec{s}) = v) \in \Gamma$, and let $D_{\simeq}$ denote the system of sets $\{[u] \cup \mathrm{G}(u)\}_{u \in D_{\mathcal{L}}}$. We prove that $D_{\simeq}$ is a partition of D: First, the union $\bigcup D_{\simeq}$ of all sets in the system is clearly a subset of D by the construction. Also, every term of $D_{\mathcal{L}}$ is in the union, and whenever $g(\vec{s}) \in D$, there is $u \in D_{\mathcal{L}}$ such that $(g(\vec{s}) = u) \in \Gamma$, whence $g(\vec{s}) \in \mathrm{G}(u)$. Therefore $\bigcup D_{\simeq} = D$.

Second, we need to prove that every two sets in the system $D_{\simeq}$ are either disjoint or equal. Note that every $\mathrm{G}(u)$ is disjoint with every $[v]$. Take any $([u] \cup \mathrm{G}(u)), ([v] \cup \mathrm{G}(v))$ in $D_{\simeq}$, and assume they are not disjoint. Then $[u] \cap [v] \neq \emptyset$, or $\mathrm{G}(u) \cap \mathrm{G}(v) \neq \emptyset$. If the former is true, then $[u] = [v]$, and consequently $\mathrm{G}(u) = \mathrm{G}(v)$ by the construction of G above. If the latter is the case, there is $g(\vec{s}) \in \mathrm{G}(u) \cap \mathrm{G}(v)$, whence $(g(\vec{s}) = u'), (g(\vec{s}) = v') \in \Gamma$ for some $u' \in [u]$, $v' \in [v]$. If u' and v' are identical terms, we have $[u] = [v]$, and $\mathrm{G}(u) = \mathrm{G}(v)$ immediately as above. Otherwise there must be a clause $(\vec{s} = \vec{s} \Rightarrow u' = v') \in \Pi$. Since $\simeq \vDash \Pi$, and obviously $\simeq \vDash \vec{s} = \vec{s}$, we have $\simeq \vDash u' = v'$, thus $[u] = [u'] = [v'] = [v]$, and again also $\mathrm{G}(u) = \mathrm{G}(v)$.

Let us now define a relation $\simeq_D$ for all $u, v \in D$ as $u \simeq_D v$ iff $u, v \in X$ for some $X \in D_{\simeq}$. Observe:

(a) $\simeq_D$ is an equivalence since $D_{\simeq}$ is a partition of D.

(b) $\simeq_D$ is a model of Γ. For every $g(\vec{s}) = u$ of Γ, we have $g(\vec{s}) \in \mathrm{G}(u) \subseteq [u]_{\simeq} \cup \mathrm{G}(u) = [u]_{\simeq_D}$, thus $\simeq_D \vDash g(\vec{s}) = u$.

(c) $\simeq$ and $\simeq_D$ coincide on $D_{\mathcal{L}}$. For any $s, t \in D_{\mathcal{L}}$, if $s \simeq t$, then $[s]_{\simeq} \cup \mathrm{G}(s) = [t]_{\simeq} \cup \mathrm{G}(t)$ as above, thus $s \simeq_D t$. Conversely, $s \simeq_D t$ implies $[s]_{\simeq} \cup \mathrm{G}(s) = [t]_{\simeq} \cup \mathrm{G}(t)$, and since $[s]_{\simeq} \cap \mathrm{G}(t) = [t]_{\simeq} \cap \mathrm{G}(s) = \emptyset$, we have $[s]_{\simeq} = [t]_{\simeq}$, whence $s \simeq t$.

(d) $\simeq_D$ is c-closed over D. Take any $f(\vec{s}), f(\vec{t}) \in D$, and assume $\simeq_D \vDash \vec{s} = \vec{t}$. If $\vec{s} \equiv \vec{t}$, then $f(\vec{s}) \simeq_D f(\vec{t})$ by reflexivity of $\simeq_D$, so assume $\vec{s} \not\equiv \vec{t}$. If f is a function symbol of $\mathcal{L}$, then $f(\vec{s}), f(\vec{t}) \in D_{\mathcal{L}}$, and $f(\vec{s}) \simeq_D f(\vec{t})$ by coincidence and $\simeq$ being a congruence. If f is some $g \in \mathcal{C}$, then still $\vec{s}, \vec{t} \in D_{\mathcal{L}}$, hence $\simeq \vDash \vec{s} = \vec{t}$ by concidence. Since $\vec{s} \not\equiv \vec{t}$, there must be a clause $(\vec{s} = \vec{t} \Rightarrow u = v) \in \Pi$ for some $u, v \in D_{\mathcal{L}}$. We obtain $\simeq \vDash u = v$ from $\simeq \vDash \Pi$, whence $g(\vec{s}) \in \mathrm{G}(u) = \mathrm{G}(v) \ni g(\vec{t})$, so $g(\vec{s}) \simeq_D g(\vec{t})$.

To sum up: $\simeq_D$ is a c-closed relation over a domain D in $D_{\mathcal{L}[\mathcal{C}]}$ satisfying Γ, and coinciding on $D_{\mathcal{L}}$ with $\simeq$. By Lemma 3.3.5, $\simeq_D$ can be expanded to a congruence $\simeq'$ over $D_{\mathcal{L}[\mathcal{C}]}$, coinciding with $\simeq_D$ on D, and consequently coinciding with $\simeq$ on $D_{\mathcal{L}}$. Therefore $\simeq'$ is a congruence satisfying Γ, thus satisfying $s = t$. From coicidence, we obtain that also $\simeq$ satisfies $s = t$, which was our original goal. □

4.1.5 An algorithm for logical consequence from sets of closed Horn clauses. When given a finite set of closed Horn clauses Π and a finite set of equalities Σ, both in a language $\mathcal{L}$, the algorithm to decide the logical consequence $\Pi \vDash \Sigma$ is as follows:

1. Build a new set of closed Horn clauses $\Pi' := (\Pi, (\Sigma \Rightarrow))$[1].
2. Transform Π' to a set of equalities Γ as in Par. 4.1.3.
3. Use the congruence closure algorithm from Sect. 3.6 to decide $\Gamma \vDash_{\mathrm{c}} \mathtt{0} = \mathtt{1}$ in the domain of the set Γ. The answer is true iff $\Pi \vDash \Sigma$ holds.

Correctness (soundness and completeness) of the algorithm is ensured by the preceding lemmas and theorem, and by correctness of the congruence closure algorithm. The first two steps are clearly linear in the total number of symbols in formulas of Π and Σ. The time required by the third step is $O(n \log n)$ where n is linear in the total length of sentences in Π, Σ.

4.2 General Clauses

We have seen in the previous section that congruence enables us to reduce Horn clauses to equalities. Intuitively, this is because congruence axioms $\vec{s} = \vec{t} \Rightarrow f(\vec{s}) = f(\vec{t})$ are also Horn clauses.

One can imagine a dual notion to a Horn clause, an "anti-Horn" clause with at most one atomic formula in the antecedent. A general clause $\Gamma \Rightarrow \Delta$ can be split up to an equivalent pair consisting of a Horn clause $\Gamma \Rightarrow c = \mathsf{o}$ and an anti-Horn clause $c = \mathsf{o} \Rightarrow \Delta$. The equality $c = \mathsf{o}$ for a new constant c links the two clauses.

We can then replace Horn clauses with pairs of equalities as in Par. 4.1.3. Anti-Horn clauses can also be replaced with equalities: For each such clause we can introduce a special function symbols h along with an anti-congruence axiom with a form dual to congruence: $h(\vec{s}) = h(\vec{t}) \Rightarrow \vec{s} = \vec{t}$. We can then replace each anti-Horn clause $c = \mathsf{o} \Rightarrow \vec{s} = \vec{t}$ with a pair of equalities $h(\vec{s}) = c$ and $h(\vec{t}) = \mathsf{o}$.

It is well known that any sequent $\Gamma \Rightarrow \Delta$ of quantifier-free sentences can be reduced to a set of general clauses Π. The sequent is valid iff the set of general clauses is unsatisfiable. As we have just seen, from the set Π, we can obtain a set of equalities equivalent to Π under anti-congruence axioms.

Our plan for this chapter is as follows: We first introduce in Sect. 4.3 anticongruence function symbols, anticongruence axioms, and relations called anticongruences, extending the semantics from Sect. 3.3. We then define in Sect. 4.4 a transformation of any closed quantifier-free sequent into a set of equalities through general clauses (but bypassing the step with anti-Horn clauses mentioned above). The transformation is similar on the one from Par. 4.1.3. We show that the transformed set is conservative over the original sequent. We then extend in Sect. 4.5 the C proof system from Par. 3.4.2 to anticongruence function symbols, and show it to be sound and complete w.r.t. the extended semantics. We describe an algorithm for congruence-anticongruence (CA) closure in Sect. 4.6.

4.3 Anticongruences and The Relation-Based Semantics

In this section, we extend the definitions and one of the lemmas from Sect. 3.3 to anticongruences.

4.3.1 Anticongruence symbols and pure languages. Let us fix a functional language $\mathcal{L}$. Some of the function symbols of positive arities in $\mathcal{L}$ can be designated as *anticongruence symbols*. We call a term in $D_{\mathcal{L}}$ *pure* if no anticongruence symbols occur in it. A *pure* language is without any anticongruence symbols. A language with anticongruence symbols is called *non-pure*.

4.3.2 Anticongruence relations and satisfiability. Fix a functional language $\mathcal{L}$ with anticongruence symbols. A congruence relation $\simeq$ is an *anticongruence* if for all anticongruence symbols $h \in \mathcal{L}$ of arity $m \geq 1$ and terms $\vec{s}$, $\vec{t}$ such that $h(s_1, \ldots, s_m) \simeq$

$h(t_1, \ldots, t_m)$, we have $s_i \simeq t_i$ for some $i = 1, \ldots, m$. Note that for pure languages, every congruence relation is trivially an anticongruence.

We define a-*model*, and a-*consequence* for formulas ($T \models_{\mathrm{a}} A$) and sets of formulas ($T \models S$) analogously to the definitions in Par. 3.3.2. Similarly, the definitions in Par. 3.3.3 can be applied to anticongruences obtaining the notions of a-*extension* and a-*conservativity*.

Let D be a domain for the language $\mathcal{L}$, and let $\simeq$ be an equivalence c-closed over D. The equivalence $\simeq$ is a-*closed* over D if for every $h(\vec{u}), h(\vec{v}) \in D$ with h an anticongruence symbol such that $h(u_1, \ldots, u_m) \simeq h(v_1, \ldots, v_m)$, we have $u_i \simeq v_i$ for some $i = 1, \ldots, m$. Similarly to e- and c-closed relations, the restriction of an anticongruence over $D_{\mathcal{L}}$ to a domain D is a-closed over D. Lemma 3.3.5 can also be formulated and proved for anticongruences.

4.3.3 Lemma (Expansion of a-closed relations). *Every* a-*closed relation $\simeq$ over a domain D of a functional language $\mathcal{L}$ can be* expanded *to an anticongruence $\simeq'$ over $D_{\mathcal{L}}$ such that $\simeq$ and $\simeq'$ coincide on D, and for all function symbols g that do not occur in the terms of D, whenever $s \in D$, and g occurs in a term t, then $\simeq' \not\models s = t$.*

Proof. The first part of the proof is identical to the proof of Lemma 3.3.5. It remains to show that $\simeq'$ defined there is an anticongruence. So assume $\mathcal{M} \models h(\vec{u}) = h(\vec{v})$ for an anticongruence symbol h of $\mathcal{L}$, and consider two cases. If $(h(\vec{u}))^{\mathcal{M}}$ is odd, then there are $\vec{s}$ and $\vec{t}$ such that $h(\vec{s}), h(\vec{t}) \in D$, $u_i^{\mathcal{M}} = \mathit{ord}([s_i])$, and $v_i^{\mathcal{M}} = \mathit{ord}([t_i])$ for all $i = 1, \ldots, m$, and $\mathit{ord}([h(\vec{s})]) = (h(\vec{u}))^{\mathcal{M}} = (h(\vec{v}))^{\mathcal{M}} = \mathit{ord}([h(\vec{t})])$. Since *ord* is an injection, we have $[h(\vec{s})] = [h(\vec{t})]$, thus $h(\vec{s}) \simeq h(\vec{t})$ from which we get $s_i \simeq t_i$ for some $i = 1, \ldots, m$, as $\simeq$ is a-closed. Hence $u_i^{\mathcal{M}} = s_i^{\mathcal{M}} = t_i^{\mathcal{M}} = v_i^{\mathcal{M}}$, and $u_i \simeq' v_i$. If $(h(\vec{u}))^{\mathcal{M}}$ is even, then $2{\cdot}u_1^{\mathcal{M}} = (h(\vec{u}))^{\mathcal{M}} = (h(\vec{v}))^{\mathcal{M}} = 2{\cdot}v_1^{\mathcal{M}}$, and hence $u_1 \simeq' v_1$. □

4.3.4 Anticongruence axioms. Fix a functional language $\mathcal{L}$ with anticongruence symbols. Similarly to congruences and sets $\mathrm{C}^{\mathcal{L}}$ in Par. 3.3.6, if $\simeq$ is an anticongruence, then it is also an a-model of the set $\mathrm{A}^{\mathcal{L}}$ of all clauses

$$h(\vec{s}) = h(\vec{t}) \Rightarrow \vec{s} = \vec{t} \tag{1}$$

where h is an anticongruence symbol of $\mathcal{L}$. Vice versa, any congruence satisfying $\mathrm{A}^{\mathcal{L}}$ is an anticongruence. Note that the set $\mathrm{A}^{\mathcal{L}}$ is empty for pure languages, and any congruence for such a language is trivially a model of $\mathrm{A}^{\mathcal{L}}$. Similarly to Thm. 3.3.7(ii), a-consequence is reducible to c-consequence under $\mathrm{A}^{\mathcal{L}}$.

4.3.5 Theorem (Reduction of a-consequence). *For each theory T in a functional language $\mathcal{L}$ and each sentence A of $\mathcal{L}$, we have $T \models_{\mathrm{a}} A$ iff $T, \mathrm{A}^{\mathcal{L}} \models_{\mathrm{c}} A$.* □

4.4 Conversion of Arbitrary Sequents to Sets of Equalities

We will transform in Par. 4.4.1 arbitrary closed quantifier-free sequents to sets of clauses without any additional connectives. In Par. 4.4.4, we will show how to elim-

inate the clauses in favor of sets of equalities with anticongruence symbols. The Conservativity Theorem 4.4.5 means that the detour through anticongruences is, strictly speaking, not necessary. However, as the simplicity of the CA proof system presented in Par. 4.5.1 and proved complete in Thm. 4.5.5 shows, it is very convenient. The soundness and completeness of our CA closure algorithm presented in Sect. 4.6 is a direct consequence of Thm. 4.5.5.

4.4.1 Transformation of sequents into a normal form. Fix a pure functional language $\mathcal{L}$. We will designate by the meta-variable Π finite sets of sequents. We say that Π is *normal* if every $\Gamma \Rightarrow \Delta$ of Π is a clause with non-empty Δ.

For a given sequent $\Gamma \Rightarrow \Delta$ in $\mathcal{L}$, we form a set containing a single sequent $(\Gamma \Rightarrow \Delta) \Rightarrow$, and repeatedly apply the following transformation rules to it:

$$
\begin{aligned}
(\Gamma \Rightarrow A \to B, \Delta), \Pi &\longrightarrow (A, \Gamma \Rightarrow B, \Delta), \Pi \\
(A \to B, \Gamma \Rightarrow \Delta), \Pi &\longrightarrow (\Gamma \Rightarrow A, \Delta), (B, \Gamma \Rightarrow \Delta), \Pi \\
(\Gamma \Rightarrow (\Gamma_1 \Rightarrow \Delta_1), \Delta), \Pi &\longrightarrow (\Gamma_1, \Gamma \Rightarrow \Delta_1, \Delta), \Pi \\
((\Gamma_1 \Rightarrow \Delta_1), \Gamma \Rightarrow \Delta), \Pi &\longrightarrow \{\Gamma \Rightarrow A, \Delta \mid A \in \Gamma_1\}, \{B, \Gamma \Rightarrow \Delta \mid B \in \Delta_1\}, \Pi \\
(\Gamma \Rightarrow A \vee B, \Delta), \Pi &\longrightarrow (\Gamma \Rightarrow A, B, \Delta), \Pi \\
(A \vee B, \Gamma \Rightarrow \Delta), \Pi &\longrightarrow (A, \Gamma \Rightarrow \Delta), (B, \Gamma \Rightarrow \Delta), \Pi \\
(A \wedge B, \Gamma \Rightarrow \Delta), \Pi &\longrightarrow (A, B, \Gamma \Rightarrow \Delta), \Pi \\
(\Gamma \Rightarrow A \wedge B, \Delta), \Pi &\longrightarrow (\Gamma \Rightarrow A, \Delta), (\Gamma \Rightarrow B, \Delta), \Pi \\
(\neg A, \Gamma \Rightarrow \Delta), \Pi &\longrightarrow (\Gamma \Rightarrow A, \Delta), \Pi \\
(\Gamma \Rightarrow \neg A, \Delta), \Pi &\longrightarrow (A, \Gamma \Rightarrow \Delta), \Pi.
\end{aligned}
$$

By the following Lemma 4.4.2, there is always a unique final set Π' for each sequent $\Gamma \Rightarrow \Delta$. The set Π' consists of clauses, i.e., sequents $\Gamma_1 \Rightarrow \Delta_1$ where Γ_1 and Δ_1 contain equalities only. Thus Π' is the *conjunctive normal form* (CNF) of the original sequent.

Let $\mathtt{1}$ be a new constant for $\mathcal{L}$, and let $\Pi := \Pi'^{\mathtt{1}}$ be as in Lemma 4.1.2. The set Π in $\mathcal{L}[\mathtt{1}]$ is normal, and we call the *normal form* of the original sequent $\Gamma \Rightarrow \Delta$.

4.4.2 Lemma. *For every set of sequents Π there is a unique set of clauses Π' such that $\Pi \longrightarrow^* \Pi'$.*

Proof. We will prove the lemma by complete induction using a measure of sets of sentences. Let us define a measure function μ from sentences to positive natural numbers by induction on the structure of sentences satisfying $\mu(s = t) = 1$; $\mu(\neg A) = \mu(A) + 1$; $\mu(A \diamond B) = 2^{\mu(A)+\mu(B)}$ for $\diamond \equiv \wedge, \vee, \to$; and $\mu(\Gamma \Rightarrow \Delta) = 2^{\mu(\Gamma,\Delta)}$ where the measure of sets of sentences $\mu(\Psi) = \sum_{A \in \Psi} \mu(A)$ for all Ψ.

We have $\mu(\Psi) > \mu(\Psi')$ whenever $\Psi \longrightarrow \Psi'$. We will only show the most complex case. The measure of the left-hand side of the fourth transformation rule is greater than the measure of its right-hand side since we have:

$$\begin{aligned}\mu((\Gamma_1 \Rightarrow \Delta_1), \Gamma \Rightarrow \Delta) &= 2^{\mu(\Gamma,\Delta)+2^{\mu(\Gamma_1,\Delta_1)}} = 2^{\mu(\Gamma,\Delta)}.2^{2^{\mu(\Gamma_1,\Delta_1)}} \\ &> 2^{\mu(\Gamma,\Delta)}.2^{\mu(\Gamma_1,\Delta_1)} \geq 2^{\mu(\Gamma,\Delta)}.\sum_{A\in\Gamma_1,\Delta_1} 2^{\mu(A)} = \sum_{A\in\Gamma_1,\Delta_1} 2^{\mu(\Gamma,\Delta)+\mu(A)} \\ &= \mu(\{\Gamma \Rightarrow A, \Delta \mid A \in \Gamma_1\}, \{B, \Gamma \Rightarrow \Delta \mid B \in \Delta_1\}).\end{aligned}$$

The first inequality follows from $2^n > n$ if $n \geq 0$, the second one from $2^{n+m} \geq 2^n + 2^m$ if $n, m > 0$, which can be extended by induction to $2^{\sum_{i=1}^k n_i} \geq \sum_{i=1}^k 2^{n_i}$ if $n_i \geq 0$ for all $i = 1, \ldots, k$.

We prove the lemma by complete induction on m for all Π such that $\mu(\Pi) = m$. Take any m and any set of sequents Π, $\mu(\Pi) = m$, and assume the claim of the second part holds for every Σ such that $m > \mu(\Sigma)$. If Π is a set of clauses, no rule can be applied to it, and the claim is immediate for $\Pi' = \Pi$. Otherwise there is a sequent $S \in \Pi$ with at least one non-atomic sentence in its antecedent or consequent, so a transformation rule is applicable.

Assume two transformation rules r_1 and r_2 can be applied to Π with different results. If the rules are applied to different sequents S_1 and S_2 respectively, there are Ψ, Π_1, and Π_2 such that $\Pi = S_1, S_2, \Psi$; $S_1 \xrightarrow{r_1} \Pi_1$; and $S_2 \xrightarrow{r_2} \Pi_2$. The two rules can be combined in any order to yield the same result: $\Pi \xrightarrow{r_1} \Pi_1, S_2, \Psi \xrightarrow{r_2} \Pi_1, \Pi_2, \Psi \xleftarrow{r_1} S_1, \Pi_2, \Psi \xleftarrow{r_2} \Pi$, and we have $m > \mu(\Pi_1, \Pi_2, \Psi)$.

Assume that both rules are applied to the same sequent $S \in \Pi$ with different results. Recall from Par. 2.3.3 that a sequent is a set of signed sentences. The rules r_1 and r_2 must apply to different signed sentences A_1 and A_2 of S respectively. We thus have $S = \{A_1, A_2\} \cup S'$ for some sequent S'; we also have $\Pi = S, \Psi$ for some Ψ. There are some Π_1 and Π_2 such that $S \xrightarrow{r_1} \Pi_1$ and $S \xrightarrow{r_2} \Pi_2$. Inspection of all rules reveals that each sequent $S_1 \in \Pi_1$ is a superset of $\{A_2\} \cup S'$, and, symmetrically, each $S_2 \in \Pi_2$ is a superset of $\{A_1\} \cup S'$. We therefore obtain the same set $\Pi_{1\times 2}$ by applying r_2 to all members of Π_1, as well as by applying r_1 to all members of Π_2. We have $m > \mu(\Pi_{1\times 2}, \Psi)$.

To sum up, if two rules are applicable to Π, there is a unique Π'' obtained by a combination of the two rules such that $\Pi \longrightarrow^* \Pi''$ and $m > \mu(\Pi'')$. This argument is easily extended to more simultaneously applicable rules. We now obtain a unique set of clauses Π' such that $\Pi'' \longrightarrow^* \Pi'$ from the induction hypothesis, and we have $\Pi \longrightarrow^* \Pi'$ as required. □

4.4.3 Theorem (Normal form of sequents). *If the sequent $\Gamma \Rightarrow \Delta$ of a pure functional language $\mathcal{L}$ without the constant* `1` *has the normal form Π, then we have $\vDash_c \Gamma \Rightarrow \Delta$ iff $\Pi \vDash_c \mathtt{0} = \mathtt{1}$.*

Proof. Assume first that the rules of transformation from Par. 4.4.1 preserve satisfiability for all congruences. Using the notation of Par. 4.4.1, we thus have $\vDash_c \Gamma \Rightarrow \Delta$ iff $((\Gamma \Rightarrow \Delta) \Rightarrow) \vDash_c (\Rightarrow)$ iff $\Pi' \vDash_c (\Rightarrow)$ iff Π' has no c-model iff $\Pi \vDash_c \mathtt{0} = \mathtt{1}$, where the last step is by Lemma 4.1.2.

To prove our assumption, we can refer to soundness and completeness of Gentzen's sequent calculus, on which the transformation rules are based. We will show here

an example of the proof for the fourth rule, which perhaps is the most complicated, and most interesting one. Proof for other rules is analogous.

So take any congruence $\simeq$ such that $\simeq \not\models ((\Gamma_1 \Rightarrow \Delta_1), \Gamma \Rightarrow \Delta), \Pi$. We prove $\simeq \not\models \{\Gamma \Rightarrow A, \Delta \mid A \in \Gamma_1\}, \{B, \Gamma \Rightarrow \Delta \mid B \in \Delta_1\}, \Pi$. That is immediate if $\simeq \not\models \Pi$. Otherwise, $\simeq \models (\Gamma_1 \Rightarrow \Delta_1), \Gamma$, but $\simeq \not\models B$ for all $B \in \Delta$. Therefore, $\simeq \models \Gamma_1 \Rightarrow \Delta_1$, and $\simeq \models \Gamma$. Consider now two cases: If $\simeq \not\models A_1$ for some $A_1 \in \Gamma_1$, then $\simeq \not\models \Gamma \Rightarrow A_1, \Delta$, which proves our goal. Otherwise $\simeq \models \Gamma_1$, and there is $B_1 \in \Delta_1$ such that $\simeq \models B_1$, whence $\simeq \not\models B_1, \Gamma \Rightarrow \Delta$.

For the opposite direction, take any congruence $\simeq$ such that $\simeq \not\models \{\Gamma \Rightarrow A, \Delta \mid A \in \Gamma_1\}, \{B, \Gamma \Rightarrow \Delta \mid B \in \Delta_1\}, \Pi$, and let us show $\simeq \not\models ((\Gamma_1 \Rightarrow \Delta_1), \Gamma \Rightarrow \Delta), \Pi$. Again, we have that immediately if $\simeq \not\models \Pi$. If there is $A \in \Gamma_1$ such that $\simeq \not\models \Gamma \Rightarrow A, \Delta$, then $\simeq \models \Gamma$, but $\simeq \not\models A$, and $\simeq \not\models B$ for all $B \in \Delta$. From the former, we have $\simeq \models \Gamma_1 \Rightarrow \Delta_1$, hence from the latter, $\simeq \not\models (\Gamma_1 \Rightarrow \Delta_1), \Gamma \Rightarrow \Delta$. Finally, if $\simeq \models \Pi$, and $\simeq \models \{\Gamma \Rightarrow A, \Delta \mid A \in \Gamma_1\}$, then there must be $B \in \Delta_1$ such that $\simeq \not\models B_1, \Gamma \Rightarrow \Delta$. Consequently $\simeq \models B_1, \Gamma$, but $\simeq \not\models B$ for all $B \in \Delta$, whence again $\simeq \models \Gamma_1 \Rightarrow \Delta_1$, and $\simeq \not\models (\Gamma_1 \Rightarrow \Delta_1), \Gamma \Rightarrow \Delta$. □

4.4.4 Conversion of normal sets of sequents to atomic theories. For a given normal set of clauses Π in a pure functional language $\mathcal{L}$, we will now construct an *associated* atomic theory Γ in a functional language $\mathcal{L}[\mathcal{A}]$ with anticongruence symbols $\mathcal{A}$. We could do that in two stages, first turning all clauses to Horn clauses using anticongruence symbols, and then employ Par. 4.1.3 to convert those into an atomic set. However, it is simpler to prove that the result of a direct conversion is a conservative extension of Π. The conversion is similar to Par. 4.1.3, and the Conservativity Theorem 4.4.5 is analogous to Thm. 4.1.4.

We construct $\mathcal{A}$ and a set of equalities Γ by the following procedure. Both sets are initially empty. The procedure takes each closed clause

$$s_1 = t_1, \ldots, s_n = t_n \Rightarrow u_1 = v_1, \ldots, u_m = v_m \tag{1}$$

in Π (note that $m \geq 1$), and

1. adds to $\mathcal{A}$ a new n-ary function symbol g and a new m-ary anticongruence symbol h;
2. adds to Γ the equalities $g(\vec{s}) = h(\vec{u})$, $g(\vec{t}) = h(\vec{v})$.

4.4.5 Theorem (Conservativity of associated atomic theories). *For every atomic theory Γ in $\mathcal{L}[\mathcal{A}]$ associated with a normal set of clauses Π in a pure $\mathcal{L}$, and for every $s = t$ in $\mathcal{L}$, we have $\Gamma \models_{\mathrm{a}} s = t$ iff $\Pi \models_{\mathrm{c}} s = t$. This is an immediate consequence of $\Gamma, \mathrm{A}^{\mathcal{L}[\mathcal{A}]}$ being c-conservative over Π for equalities.*

Proof. We first prove that $\Gamma, \mathrm{A}^{\mathcal{L}[\mathcal{A}]}$ is a c-extension of Π by showing $\Gamma, \mathrm{A}^{\mathcal{L}[\mathcal{A}]} \models_{\mathrm{c}} \Pi$, i.e., $\Gamma \models_{\mathrm{a}} \Pi$. So take any clause 4.4.4(1) of Π and any a-model $\simeq$ of $\Gamma, \vec{s} = \vec{t}$. Thus $\simeq \models g(\vec{s}) = g(\vec{t})$, and hence $\simeq \models h(\vec{u}) = h(\vec{v})$. But then $\simeq \models u_i = v_i$ for some $i = 1, \ldots, m$.

We now prove that $\Gamma, \mathrm{A}^{\mathcal{L}[\mathcal{A}]}$ is c-conservative over Π for equalities. So assume $\Gamma \models_{\mathrm{a}} s = t$ for an equality $s = t$ in $\mathcal{L}$. We wish to prove $\Pi \models_{\mathrm{c}} s = t$. Take any c-model $\simeq$ of Π. Let $\simeq_D$ be the restriction of $\simeq$ to the domain D of $\Pi, s = t$. Clearly, $\simeq_D$ is c-closed. Note that the domain D^{a} of $\Gamma, s = t$ contains, in addition to the terms of D, also the four terms $g(\vec{s})$, $h(\vec{u})$, $g(\vec{t})$, $h(\vec{v})$ for each clause 4.4.4(1) of Π. Take the set $\simeq_D \cup \{\langle g(\vec{s}), h(\vec{u})\rangle \mid (g(\vec{s}) = h(\vec{u})) \in \Gamma\}$, and extend it to the least c-closed equivalence $\simeq_{D^{\mathrm{a}}}$ over D^{a}. For all clauses 4.4.4(1) in Π, this happens by having $[g(\vec{s})]_{\simeq_{D^{\mathrm{a}}}} = \{g(\vec{s}), h(\vec{u}), g(\vec{t}), h(\vec{v})\}$ when $\simeq \models \vec{s} = \vec{t}$ or $\simeq \models \vec{u} = \vec{v}$ holds. Otherwise we have $[g(\vec{s})]_{\simeq_{D^{\mathrm{a}}}} = \{g(\vec{s}), h(\vec{u})\}$, and $[g(\vec{t})]_{\simeq_{D^{\mathrm{a}}}} = \{g(\vec{t}), h(\vec{v})\}$. Note that $\simeq_D = \simeq_{D^{\mathrm{a}}} \cap (D \times D)$.

We now show that $\simeq_{D^{\mathrm{a}}}$ is a-closed. If $h(\vec{u}) \simeq_{D^{\mathrm{a}}} h(\vec{v})$ for the anticongruence symbol h associated with the clause 4.4.4(1), then there are two cases. Either $\simeq \models \vec{s} = \vec{t}$, and then, since $\simeq$ is a c-model of Π, we have $\simeq \models u_i = v_i$ for some $i = 1, \ldots, m$. From this we get $u_i \simeq_{D^{\mathrm{a}}} v_i$. Or else $\simeq \models \vec{u} = \vec{v}$, i.e., $u_1 \simeq_D v_1$, and hence $u_1 \simeq_{D^{\mathrm{a}}} v_1$.

By Lemma 4.3.3, there is an anticongruence $\simeq'$ over $D_{\mathcal{L}[\mathcal{A}]}$ which is an expansion of $\simeq_{D^{\mathrm{a}}}$. By the construction of $\simeq_{D^{\mathrm{a}}}$, the relation $\simeq'$ is an a-model of Γ. From the assumption $\Gamma \models_{\mathrm{a}} s = t$, we get $\simeq' \models s = t$, and hence $\simeq \models s = t$ because $\simeq$ and $\simeq'$ coincide on D. □

4.4.6 Practical considerations. Note that the size of the set Π' produced by the transformation given in Par. 4.4.1 can be exponential in the size of the original sequent $\Gamma \Rightarrow \Delta$. Using this transformation has simplified the proof of Thm. 4.4.3. But a practical application should use a different transformation which yields only polynomial increase in size of the original sequent. This is possible, at the expense of introduction of new constants, using an adapted version of the transformation to the so called *equisatisfiable* CNF, which, according to [DNS05], can be traced back to Skolem.

We introduce a new constant c_A for every non-atomic subformula of the sequent $(\Gamma \Rightarrow \Delta) \Rightarrow$. For each subformula A of $(\Gamma \Rightarrow \Delta) \Rightarrow$, we define a *proxy* $\mathrm{P}(A)$ *for* A so that $\mathrm{P}(A) :\equiv A$ if A is atomic, and $\mathrm{P}(A) :\equiv (c_A = \mathtt{o})$ if A is non-atomic. The set Π' is then the union of $\{\mathrm{P}((\Gamma \Rightarrow \Delta) \Rightarrow)\}$ and all $\Pi(A)$ for non-atomic subformulas A of $(\Gamma \Rightarrow \Delta) \Rightarrow$. The set of clauses $\Pi(A)$ defines $\mathrm{P}(A)$ as equivalent to A. Specifically, for any A, B, Γ', Δ', we have

(i) $\Pi(\neg A) = (\mathrm{P}(A), \mathrm{P}(\neg A) \Rightarrow), (\Rightarrow \mathrm{P}(A), \mathrm{P}(\neg A))$;

(ii) $\Pi(A \wedge B) = (\mathrm{P}(A \wedge B) \Rightarrow \mathrm{P}(A)), (\mathrm{P}(A \wedge B) \Rightarrow \mathrm{P}(B)),$
$(\mathrm{P}(A), \mathrm{P}(B) \Rightarrow \mathrm{P}(A \wedge B))$;

(iii) $\Pi(A \vee B) = (\mathrm{P}(A \vee B) \Rightarrow \mathrm{P}(A), \mathrm{P}(B)),$
$(\mathrm{P}(A) \Rightarrow \mathrm{P}(A \vee B)), (\mathrm{P}(B) \Rightarrow \mathrm{P}(A \vee B))$;

(iv) $\Pi(A \to B) = (\mathrm{P}(A), \mathrm{P}(A \to B) \Rightarrow \mathrm{P}(B)),$
$(\Rightarrow \mathrm{P}(A), \mathrm{P}(A \to B)), (\mathrm{P}(B) \Rightarrow \mathrm{P}(A \to B))$;

(v) $\Pi(\Gamma' \Rightarrow \Delta') = (\mathrm{P}(\Gamma'), \mathrm{P}(\Gamma' \Rightarrow \Delta') \Rightarrow \mathrm{P}(\Delta')),$
$\{(\Rightarrow \mathrm{P}(A), \mathrm{P}(\Gamma' \Rightarrow \Delta')) \mid A \in \Gamma'\},$
$\{(\mathrm{P}(B) \Rightarrow \mathrm{P}(\Gamma' \Rightarrow \Delta')) \mid B \in \Gamma'\}$

where $\mathrm{P}(\Sigma) = \{\mathrm{P}(A) \mid A \in \Sigma\}$ for any Σ.

Furthermore, any practical application of the CA closure algorithm will use the following obvious simplifications of the conversion process described in Par. 4.4.4. We did not use them in order to keep the exposition simple. Whenever 4.4.4(1) is a Horn clause, i.e., $m = 1$, there is no need to introduce a new anticongruence symbol h into $\mathcal{L}[\mathcal{A}]$. The addition of the equalities $g(\vec{s}) = u_1$ and $g(\vec{t}) = v_1$ into Γ suffices (however, a comparison with Thm. 4.1.4 reveals that leaving out this optimization has notably simplified the proof of Thm. 4.4.5). If we have $n = 0$, there is no need to introduce the constant g into $\mathcal{L}[\mathcal{A}]$. The addition of $h(\vec{u}) = h(\vec{v})$ into Γ suffices for $m > 1$. If $m = 1$, we just add $u_1 = v_1$.

Thus if Π consists of Horn clauses, then the CA proof system is without the 4.5.1(3) rules, and a congruence closure algorithm suffices to decide whether $s = t$ is a c-consequence of Π as in Sect. 4.1.

For the sake of simplicity, we will ignore below the optimizations described in this paragraph.

4.5 Congruence-Anticongruence Proof System

As we have announced in the Overview Section 4.2, we extend the C proof system of Par. 3.4.2 to deal with anticongruence symbols in this section.

4.5.1 The CA proof system. We will now present a proof system for proving a-consequences of atomic sets Γ. The reader will note that the system restricts the use of equality axioms from the sets $\mathrm{C}^{\mathcal{L}}$, $\mathrm{A}^{\mathcal{L}}$ to a pure domain, and to non-pure terms occurring in Γ.

For any functional language $\mathcal{L}$ with anticongruence symbols, any pure domain D in $\mathcal{L}$, and any atomic set Γ such that all pure terms occurring in Γ are in D, the rules of derivation of the *CA proof system* are the following ones:

$$\frac{(\Rightarrow f(\vec{s}) = f(\vec{t})), \Gamma}{\Gamma} \quad \text{if } \Gamma^{\mathrm{e}} \models_{\mathrm{e}} \vec{s} = \vec{t}, \text{ and } f(\vec{s}), f(\vec{t}) \in D; \quad (1)$$

$$\frac{(s = t), \Gamma}{\Gamma} \quad \text{if } (\Rightarrow s = t) \in \Gamma; \quad (2)$$

$$\frac{(\Rightarrow u_1 = v_1), \Gamma \mid \cdots \mid (\Rightarrow u_m = v_m), \Gamma}{\Gamma} \quad \begin{array}{l}\text{if } h \text{ is an anticongruence symbol,} \\ h(\vec{u}) \text{ and } h(\vec{v}) \text{ occur in } \Gamma, \text{ and} \\ \Gamma^{\mathrm{e}} \models_{\mathrm{e}} h(\vec{u}) = h(\vec{v}).\end{array} \quad (3)$$

A *CA derivation for an atomic set* Γ is a finite tree built by the rules of derivation of the CA proof system for the root Γ. By induction on construction of CA derivations, we can easily prove that every node in a CA derivation is an atomic theory,

which is a superset of its parent (if there is one) and thus of the root, all pure terms occurring in a node are from D, and all non-pure equalities occurring in a node occur in the root (possibly in atomic clauses). We call a CA derivation *strictly monotonic* if every non-root node is a proper superset of its parent.

We write $\Gamma \vdash_{\mathrm{CA}} s = t$ if there is a CA derivation for Γ such that in every leaf Λ, we have $\Lambda^{\mathrm{e}} \models_{\mathrm{e}} s = t$.

4.5.2 Saturation, canonical derivations. We define saturation of an atomic theory w.r.t. a derivation rule of the CA proof system as in Par. 3.4.3. We also reuse the definition of weakly C-saturated theory. We call an atomic theory Γ *weakly CA-saturated* if it is saturated w.r.t. the derivation rule 4.5.1(3), and it is weakly C-saturated. A CA derivation is *canonical* if the premise of every application of the rule 4.5.1(3) is a weakly C-saturated theory.

4.5.3 Lemma. *Let $\mathcal{L}$ be a language with anticongruence symbols, and D a domain in the language. For any atomic theory Γ in $\mathcal{L}$ with all pure terms in D there exists a strictly monotonic canonical CA derivation in which all leaves are weakly CA-saturated.*

Proof. Take any $\mathcal{L}$, D, and Γ meeting the assumptions of the lemma. Let us denote the union of D and the domain of Γ (including non-pure terms) by D'. Similarly to the proof of Lemma 3.4.4(ii), we abbreviate by $\Gamma_{D'}$ the set $\bigcup_{s,t\in D'}\{s = t, (\Rightarrow s = t)\}$. We can prove by induction on construction the following properties: every node of every CA derivation for Γ is a subset of $\Gamma_{D'}$; the length of a path through a strictly monotonic CA derivation for Γ is at most $|\Gamma_{D'}|$; the size of such a derivation is at most $S := \max(2, M)^{|\Gamma_{D'}|}$, where M is the maximum arity of anticongruence symbols occurring in D'.

We show by induction on k that every strictly monotonic canonical CA derivation for Γ of the size s such that $S - s < k$ can be extended to a CA derivation with all leaves weakly CA-saturated. The lemma follows by using this auxiliary proposition for the derivation containing only the root Γ and $k = S$.

There is nothing to prove in the basis of the induction since $S - s$ is non-negative. In the inductive step, take any strictly monotonic canonical CA derivation for Γ of size s such that $S - s < k+1$, and assume the proposition holds for k. If $S - s < k$, the inductive step follows immediately by the induction hypothesis. Otherwise $S - s = k$. If $k = 0$, all leaves must be $\Gamma_{D'}$, which contains all possible consequences of all rules. Assume now $k > 0$. If all leaves are weakly CA-saturated, we are done. Otherwise if there is a leaf Λ not saturated w.r.t. 4.5.1(1), then we obtain a new CA derivation by applying the rule for that atomic clause $(\Rightarrow f(\vec{s}) = f(\vec{t}))$, $f(\vec{s}), f(\vec{t}) \in D$, which is not in Λ. The new derivation is strictly monotonic and canonical, and its size is $s + 1$, thus the induction hypothesis applies. We proceed similarly in case there is a leaf not saturated w.r.t. the rule 4.5.1(2).

If all leaves are saturated w.r.t. both 4.5.1(1) and 4.5.1(2) (and therefore weakly C-saturated), there must be some not saturated w.r.t. 4.5.1(3). Take one such leaf Λ,

and obtain a new CA derivation by applying the rule 4.5.1(3) to Λ for some terms $h(\vec{u})$ and $h(\vec{v})$ occurring in Λ with h an anticongruence symbol of some arity $m > 0$, and we have $\Lambda^{\mathrm{e}} \models_{\mathrm{e}} h(\vec{u}) = h(\vec{v})$, but $(\Rightarrow u_i = v_i) \notin \Lambda$ for every $i = 1, \ldots, m$. The new derivation is clearly strictly monotonic and canonical, and its size is $s + m > s$. Since the size of every strictly monotonic derivation for Γ is at most S, we have $S \geq s + m$, whence $k = S - s \geq m$, and $k = S - s > S - (s + m) \geq 0$. Thus the induction hypothesis applies again. □

4.5.4 Lemma. *Let $\mathcal{L}$ be a pure functional language, and D a domain in $\mathcal{L}$. Furthermore, let Π be a normal set of sequents containing only the terms of D, and let Γ be its associated atomic theory Γ in $\mathcal{L}[\mathcal{A}]$. For every CA derivation for Γ with weakly CA-saturated leaves, and any equality $s = t$ in the domain D, if we have $\Gamma \models_{\mathrm{a}} s = t$, then $\Lambda^{\mathrm{e}} \models_{\mathrm{e}} s = t$ in all leaves Λ of the derivation.*

Proof. Take any $\mathcal{L}$, D, Π, $\mathcal{L}[\mathcal{A}]$, and Γ as required by the assumptions of the lemma, a CA derivation for Γ with weakly CA-saturated leaves, and some terms $s, t \in D$, and assume that there is a leaf Λ in the derivation, such that $\Lambda^{\mathrm{e}} \not\models_{\mathrm{e}} s = t$. We wish to prove $\Gamma \not\models_{\mathrm{a}} s = t$.

Let $\simeq$ be the restriction of the equivalence $\simeq_{\Lambda^{\mathrm{e}}}$ to the domain D of Π. The relation is clearly c-closed and trivially a-closed because there are no anticongruence symbols in D. We expand $\simeq$ by Lemma 4.3.3 to a (trivial) anticongruence $\simeq'$ over $D_{\mathcal{L}}$.

In order to prove $\simeq' \models \Pi$, take any clause 4.4.4(1) in Π, and assume $\simeq' \models \vec{s} = \vec{t}$. Then $\vec{s} \simeq \vec{t}$, and hence $\vec{s} \simeq_{\Lambda^{\mathrm{e}}} \vec{t}$. Since Λ is weakly CA-saturated, we obtain $g(\vec{s}) \simeq_{\Lambda^{\mathrm{e}}} g(\vec{t})$. Since $(g(\vec{s}) = h(\vec{u})), (g(\vec{t}) = h(\vec{v})) \in \Gamma \subseteq \Lambda$, we also get $h(\vec{u}) \simeq_{\Lambda^{\mathrm{e}}} h(\vec{v})$. Thus $u_i \simeq_{\Lambda^{\mathrm{e}}} v_i$, and hence $\simeq' \models u_i = v_i$, for some $i = 1, \ldots, m$.

From $\Lambda \not\models_{\mathrm{e}} s = t$, we get $s \not\simeq_{\Lambda^{\mathrm{e}}} t$, then $s \not\simeq t$, and hence $\simeq' \not\models s = t$, all by coincidence. This shows $\Pi \not\models_{\mathrm{c}} s = t$, and hence $\Gamma \not\models_{\mathrm{a}} s = t$ by the Conservativity Thm. 4.4.5. □

4.5.5 Theorem (Soundness and completeness of the CA proof system for associated sets). *Let $\mathcal{L}$ be a pure functional language, and D a domain in $\mathcal{L}$. For every normal set of sequents Π containing only the terms of D, its associated atomic theory Γ in $\mathcal{L}[\mathcal{A}]$, and any equality $s = t$ in the domain D, we have $\Gamma \vdash_{\mathrm{CA}} s = t$ iff $\Gamma \models_{\mathrm{a}} s = t$.*

Proof. Actually, we prove by induction on the derivations of $\Gamma \vdash_{\mathrm{CA}} s = t$ that the CA proof system is sound for arbitrary atomic theory Γ in some $\mathcal{L}$. If the proof is a leaf, then we get $\Gamma \models_{\mathrm{a}} s = t$ because $\Gamma^{\mathrm{e}} \models_{\mathrm{e}} s = t$. If the last rule used in the proof is 4.5.1(1), then we have $\Gamma^{\mathrm{e}} \models_{\mathrm{e}} \vec{s} = \vec{t}$, and we take any a-model $\simeq$ of Γ. Since $\simeq \models \vec{s} = \vec{t}$, we get $\simeq \models f(\vec{s}) = f(\vec{t})$, and then $\simeq \models s = t$ from the inductive hypothesis. If the last rule used in the proof is 4.5.1(2), every a-model $\simeq$ of Γ is an a-model of $(s = t), \Gamma$, and $\simeq \models s = t$ by IH. If the last rule used in the proof is 4.5.1(3), then we have $\Gamma^{\mathrm{e}} \models_{\mathrm{e}} h(\vec{u}) = h(\vec{v})$, and we take any a-model $\simeq$ of Γ. Thus $\simeq \models h(\vec{u}) = h(\vec{v})$, and hence $\simeq \models u_i = v_i$ for some $i = 1, \ldots, m$. But then $\simeq \models s = t$ by the i-th inductive hypothesis.

For the proof of completeness restricted to the associated sets, take now the Γ associated with Π from the statement of the theorem. By Lemma 4.5.3, there is a CA derivation for Γ with weakly CA-saturated leaves. Now for any equality $s = t$ in the domain D, if $\Gamma \models_{\mathrm{a}} s = t$, then $\Lambda^{\mathrm{e}} \models_{\mathrm{e}} s = t$ in all leaves Λ, hence the derivation is a witness of $\Gamma \vdash_{\mathrm{CA}} s = t$. □

4.6 An Algorithm for the CA closure

We will now present an algorithm which decides $\Pi \models_{\mathrm{c}} s = t$ for a normal set Π, by deciding $\Gamma \models_{\mathrm{a}} s = t$ for the atomic theory Γ associated with Π. The working of the algorithm is similar to the proof of Lemma 4.5.3. The congruence closure algorithm from Chapter 3 is used instead of rules 4.5.1(1, 2). Branching of derivation caused by the rule 4.5.1(3) is handled by recursive calls. Since propositional satisfaction is clearly reducible to the decision problem solved by the algorithm, the decision problem is NP-complete, and our algorithm is exponential.

We fix for the rest of this section a pure functional language $\mathcal{L}$, a domain D in $\mathcal{L}$, a normal set of sequents Π with all terms in D, and its associated atomic theory Γ in a language $\mathcal{L}[\mathcal{A}]$.

4.6.1 Set-theoretical CA closure algorithm. We denote by D' the union of D with the domain of Γ (as in the proof of Lemma 4.5.3). The only terms in D' that are not already in D are all $g(\vec{s})$ and $h(\vec{u})$ associated with clauses of Π. Let us denote by D_{p} the pure part of D', which consists of the terms of D and the terms $g(\vec{s})$. These designations will also be fixed in this section.

Note that all equalities in Γ contain anticongruence symbols. They are thus ignored by the "C part" of the CA proof system, i.e., the rules 4.5.1(1, 2). As noted in Par. 4.5.1, each node Δ in a CA derivation for Γ is a superset of Γ, and all non-pure equalities in Δ are already in Γ. Therefore, Γ is the non-pure part of Δ, and $\Delta \setminus \Gamma$ is the pure part of Δ, containing only equalities among the terms of D_{p}.

Recall that we found out in Par. 3.5.13 that for a (pure) atomic theory Γ, there is exactly one tree with no sibling leaves, $T(\mathrm{C}^*(\Gamma))$, reachable by iterated application of the function *Join_classes* to pairs of labels of sibling leaves. Moreover, the tree for every weakly C-saturated theory Γ' derivable from Γ is also equal to $T(\mathrm{C}^*(\Gamma))$.

We design the algorithm for deciding a-consequence of a pure equality $s = t$ from Γ by combining above observations with properties of canonical CA derivations. The algorithm is as follows, with Δ initialized to Γ:

1. Construct the tree $T(\Delta \setminus \Gamma)$ in the domain D_{p}, and iteratively apply the function *Join_classes* to obtain the tree $T(\Delta') = T(\mathrm{C}^*(\Delta \setminus \Gamma))$.
2. Return **true** if there is a leaf in $T(\Delta')$ whose label contains both s and t.
3. Take some pair of equalities $g(\vec{s}) = h(\vec{u})$ and $g(\vec{t}) = h(\vec{v})$ from Γ satisfying:
   (i) there is a leaf in $T(\Delta')$ whose label contains both $g(\vec{s})$ and $g(\vec{t})$;
   (ii) for all $i = 1, \ldots, m$, no label of any leaf in $T(\Delta')$ contains both u_i and v_i.

If there is no such pair of equalities, return **false**.

4. For all $i = 1, \ldots, m$, apply the algorithm recursively to the atomic theory $(u_i = v_i), \Delta, \Delta'$ in place of Δ. Return the conjunction of all recursive calls.

We prove below the correctness of the set-theoretic algorithm. The design of an imperative version of the algorithm is straightforward, we will not go into technical details.

4.6.2 Theorem (Correctness of the CA closure algorithm). *For every equality $s = t$ in the domain D, the CA closure algorithm terminates, returning* **true** *or* **false**. *It returns* **true** *iff $\Gamma \models_{\mathrm{a}} s = t$.*

Proof. The algorithm mimics the construction from the proof of Lemma 4.5.3. We will prove the following claim:

Let $\Delta \supseteq \Gamma$ be such that $\Delta \setminus \Gamma$ is pure and does not contain equalities $g(\vec{s}) = t$ between any $t \in D$ and any $g(\vec{s})$ associated with some clause of Π. If the algorithm starts with Δ, it terminates returning **true** *or* **false**, *and there is a CA derivation from Δ with weakly CA-saturated leaves such that the algorithm returns* **true** *iff $\Lambda^{\mathrm{e}} \models_{\mathrm{e}} s = t$ in every leaf Λ of the derivation.*

For Γ in place of Δ, we will then have termination, and if the algorithm returns **true**, then $\Gamma \models_{\mathrm{a}} s = t$ by soundness of the CA proof system [Thm. 4.5.5($\to$)], and if the algorithm returns **false**, then $\Gamma \not\models_{\mathrm{a}} s = t$ by Lemma 4.5.4.

Let us denote by $\mu(\Delta)$ the number in Δ of pairs of equalities $g(\vec{s}) = h(\vec{u})$, $g(\vec{t}) = h(\vec{v})$ associated with some clause 4.4.4(1) of Π such that $\Delta^{\mathrm{e}} \not\models_{\mathrm{e}} u_i = v_i$ for all $i = 1, \ldots, m$. We prove the claim made above by induction on n for all Δ such that $\mu(\Delta) \leq n$.

Take any Δ satisfying the requirements of the above claim. Regardless of $\mu(\Delta)$, we can construct a C derivation starting with $\Delta \setminus \Gamma$ and ending in a weakly C-saturated atomic theory Δ' by Lemma 3.4.4(ii). By Par. 3.5.13, $T(\Delta') = T(\mathrm{C}^*(\Delta \setminus \Gamma))$. By unioning each element of the derivation with Γ, we obtain a CA derivation applying only the rules 4.5.1(1, 2), and ending with the weakly C-saturated set Δ', Γ. One can also easily prove that there are no equalities $g(\vec{s}) = t$ in Δ' with $t \in D$ and $g(\vec{s})$ associated with a clause of Π.

In the basis of the induction, we have $\mu(\Delta) \leq n = 0$, therefore both Δ and Δ', Γ are 4.5.1(3)-saturated, and consequently Δ', Γ is weakly CA-saturated. Thus we have a CA derivation from Δ with a single leaf which is weakly CA-saturated. If there is a leaf in $T(\Delta')$ whose label contains both s and t, then the algorithm returns **true**, and at the same time, we have $(\Delta')^{\mathrm{e}} \models_{\mathrm{e}} s = t$ by Lemma 3.5.4(i), thus also $(\Delta', \Gamma)^{\mathrm{e}} \models_{\mathrm{e}} s = t$. Otherwise the algorithm returns **false** since by the assumption $\mu(\Delta) = 0$, there are no pairs of equalities required in the step 3 of the algorithm. No label of any leaf in $T(\Delta')$ contains both s and t, thus we have $(\Delta')^{\mathrm{e}} \not\models_{\mathrm{e}} s = t$ by Lemma 3.5.4(i). It cannot be the case that $(\Delta', \Gamma)^{\mathrm{e}} \models_{\mathrm{e}} s = t$. This is because there are no equalities $g(\vec{s}) = t$ with $t \in D$ in Δ', Γ, thus we can construct an equivalence satisfying $(\Delta', \Gamma)^{\mathrm{e}}$ with equivalence classes $[g(\vec{s})]$, $[h(\vec{u})]$ not containing any terms of D. Therefore $(\Delta', \Gamma)^{\mathrm{e}} \not\models_{\mathrm{e}} s = t$.

In the inductive step, assume $\mu(\Delta) \leq n+1$. In case the algorithm returns **true** in step 2, we have $(\Delta')^{\mathrm{e}} \models_{\mathrm{e}} s = t$ as above, and we can construct a canonical CA derivation with weakly CA-saturated leaves for Δ', Γ by Lemma 4.5.3. We will then have $\Lambda^{\mathrm{e}} \models_{\mathrm{e}} s = t$ in each leaf $\Lambda \supseteq (\Delta', \Gamma) \supseteq \Delta'$. If the algorithm proceeds to step 3, we continue as in the basis if there are no required pairs of equalities. Otherwise the algorithm picks a pair of equalities $g(\vec{s}) = h(\vec{u})$ and $g(\vec{t}) = h(\vec{v})$ associated with a clause in Π and satisfying the requirements (i) and (ii). By Lemma 3.5.4(i), we have $(\Delta')^{\mathrm{e}} \not\models_{\mathrm{e}} u_i = v_i$ for all $i = 1, \ldots, m$. Consequently $(\Delta', \Gamma)^{\mathrm{e}} \not\models_{\mathrm{e}} u_i = v_i$ for the same reasons as in case of returning **false**. Therefore $\mu((u_i = v_i), \Delta', \Gamma) < \mu(\Delta) \leq n+1$, thus $\mu((u_i = v_i), \Delta', \Gamma) \leq n$, for all $i = 1, \ldots, m$. By the induction hypothesis for every $i = 1, \ldots, m$, the i-th recursive call terminates and there is a CA derivation with weakly CA-saturated leaves for $(u_i = v_i), \Delta', \Gamma$. We add these derivations as branches to the initial CA derivation from Δ to Δ', Γ, thus obtaining a CA derivation from Δ with weakly CA-saturated leaves. If all recursive calls return **true**, **true** is returned in step 4 also for Δ, and at the same time we have $\Lambda^{\mathrm{e}} \models_{\mathrm{e}} s = t$ in every leaf Λ of the derivation from Δ. If some recursive call returns **false**, the algorithm returns **false** also for Δ, and at the same time we have $\Lambda^{\mathrm{e}} \not\models_{\mathrm{e}} s = t$ in some leaf Λ of the derivation from Δ. □

4.7 Related Work

The results of the present chapter are related to areas of automatic theorem proving called SAT and SMT solving. We will describe these areas in the following two paragraphs, and then comment on the relationship to our work.

4.7.1 DPPL-based SAT solving. Decision of satisfiability of a set of propositional formulas, known as *SAT solving*, is a well-known NP-complete problem. It is present in automated deduction in any richer logic. Also, many practical problems can be straightforwardly reduced to SAT solving. The most prominent of them is the verification (model checking) of finite-state systems, such as computer hardware, concurrent processes, and communication protocols. SAT solving is also used for, e.g., deduction in various specialized weak logics, or computation in answer-set programming.

Since the problem is NP-complete, all known deterministic SAT solving algorithms have exponential time complexity in the worst case. Currently, the most successful method is based on the propositional part of an early automated first-order deduction method of [DLL62] which modified the one of [DP60]. The method came to be known as the *Davis-Putnam-Logemann-Loveland (DPLL) algorithm*. An important advantage of the DPLL algorithm over other methods (such as BDDs, popular in 1990s) is linear space complexity. The interest in the algorithm was revived in late 1990s, and an important milestone in its evolution is the SAT solver Chaff ([MMZ+01]).

We follow the survey [DGP04] to describe the core algorithm of modern high-performance DPLL-style SAT solvers. The solvers work with formulas in the conjunctive normal form, i.e., sets of clauses. The algorithm searches for a satisfying assignment

of truth values to propositional atoms by maintaining a partial assignment, which is initially empty, and updating it as follows:

(i) Starting with a partial assignment P, its extension P' is produced by *unit propagation* (a kind of constraint propagation): For each clause which contains exactly one literal unvalued in P (a unit clause), the assignment is extended so that the clause is satisfied.

(ii) If a clause is falsified by P', unsatisfiability in P is reported. If P' is a total assignment, the set of clauses is satisfiable, and P' is a witness.

(iii) Otherwise, a literal L unvalued in P' is chosen for a *case split*: The procedure is recursively called first with P' extended to satisfy L, and then with P' extended to falsify L. The set of clauses is satisfiable iff any recursive call reports satisfiability.

Modern DPLL-based solvers achieve efficiency by a combination of careful implementation and enhancements of the basic algorithm. Unit propagation substantially reduces the number of partial assignments considered, and solvers tend to spend considerable amount of time finding unit clauses. A successful technique limiting the number of clauses examined is known as *two watched literals*.

An important enhancement of the basic DPLL algorithm is *lemma learning*. A lemma is any clause implied by the original set of clauses. It can thus be added to the set without affecting its satisfiability. Lemmas pose additional constraints on partial assignments through unit propagation, thus pruning the search space. Whenever unsatisfiability is detected, auxiliary information maintained by the solver enables it to identify a subset of the current partial assignment which causes unsatisfiability, and generate a lemma that will make it avoid the subset in future. The information maintained for lemma learning also allows *backjumping*, i.e., non-chronological backtracking to a case split other than the most recent one. Since the number of lemmas can grow exponentially, the solver must forget lemmas. The selection of lemmas to learn and to forget is governed by heuristic criteria, mostly based on the length of the lemma and its usage in the recent history of the search.

Finally, the choice of a literal for the case split in step (iii) of the algorithm can have a great impact on the performance of the solver. Several heuristics exist for this task. Learning solvers tend to favor case splits on literals in recently learned lemmas.

Much work is invested into scientific engineering of SAT solvers, i.e., into implementation using efficient data structures, rather low-level optimizations, and heuristics, and into tuning solvers for specific needs of various applications (e.g., [ES03]). For several classes of problems (pigeonhole, parity, clique coloring), the smallest proof in resolution-based proof systems (including the one implemented by DPLL SAT solvers) is exponentially large, but there are other proof systems admitting polynomial proofs of these problems. Finding DPLL-like algorithms for such proof systems (e.g., [DGLP04]) is one of the less engineering-oriented efforts.

A common theoretical framework for variations and extensions of the DPLL algorithm was proposed in [NOT05], abstracting the algorithm into a state transition system. With the same purpose, we have described a framework based on the standard logical method of forcing in [KV05].

4.7.2 Satisfiability modulo theories. The success of SAT solving in computer-aided electronic design drives research into its applications to more complex verification tasks: As hardware complexity grows, its specification in more expressive logics, and direct verification against this specification is desirable. Also, while full software verification is a too complex goal, *static checking*, i.e., identification of common programming errors which are beyond most type-checking systems, appears feasible. This includes array and other bounds checking, dereferencing of null pointer or reference, type checking of classes, such as arrays, vectors, sets, which would be parameterized by type if the language allowed it.

These tasks can often be reduced to decision of satisfiability of a set of clauses with equality in presence of a theory T (*satisfiability modulo* T, SMT), which is usually a combination of several "small theories" such as the basic theory of equality with uninterpreted function symbols (EUF; equivalent to our c-consequence), linear integer and real arithmetic, theories of recursive data structures (one of them is the theory of Lisp-like lists), arrays, finite sets and multisets, partial and total orders, etc. Each of these small theories has its own decision procedure, which can be combined into the decision procedure for T using, e.g., the Nelson–Oppen combination method ([NO79, TH96]).

In the most successful, *lazy* approach to satisfiability modulo T, the SAT solver treats the atomic formulas in the original problem as propositional variables. Unless the problem is propositionally unsatisfiable, the solver will produce a satisfying assignment of truth values to the atomic formulas. The assignment may be inconsistent with T, for instance, an assignment in which $x = y$ is true and $f(x) = f(y)$ is false is inconsistent with the theory of equality. The assignment is passed to a theory-specific decision procedure which determines its consistency with T. If the procedure finds the assignment inconsistent, the solver is asked to produce another one.

There is a more efficient variation of the lazy cooperation schema, in which the theory-specific procedure *explicates the proof* of unsatisfiability of an assignment, i.e., provides the solver with a lemma or lemmas to avoid a T-inconsistent subset of the assignment in the subsequent search. Another variation is the *non-exhaustive eager* cooperation schema, where the solver informs the theory-specific procedure of each change in the partial truth assignment, and the procedure reports back a set of lemmas even when the partial assignment is consistent, in order to constrain its future extension by the solver.

Some variations in the cooperation schema are compared in [FJOS03]. An architecture named DPLL(T) for an non-exhaustive eager schema defined by an interface between the solver and the theory-specific procedure is described in [GHN04], and an abstract view of the architecture is given in the paper [NOT05] and (to a limited extent) in our [KV05], both mentioned already in Par. 4.7.1. The DPLL(T) framework is implemented in BarcelogicTools ([NO05]). A non-exhaustive eager SMT solver with a more traditional SAT-solving core is implemented in the theorem prover Simplify ([DNS05]).

4.7.3 Our contribution. We have shown how to absorb the propositional content of formulas into sets of equalities where the congruence axioms take care of conjunctions, the anticongruence axioms of disjunctions. Negations $\neg A$ are translated to $A \Rightarrow \mathtt{0} = \mathtt{1}$, and sequents are taken care of by equalities connecting congruence with anticongruence symbols. Of course, it is well-known in logic that formulas can be reduced to equalities. For instance, the primitive recursive arithmetic (PRA) can be presented in this way. However, one needs a certain amount of arithmetic for this. We have shown that the reduction is possible without any arithmetic by employing in a pleasingly symmetric way congruences and anticongruences.

We took the detour through anticongruences in order to have a proof system which can be directly transformed to a generalization of the congruence closure algorithm. For applications such as SAT solving, where the decision of formulas with equalities is not deemed essential, the detour can be eliminated as follows. In order to decide tautologically a sequent $\Gamma \Rightarrow \Delta$ in an arbitrary first-order language (not only a functional one), we use the transformations in Par. 4.4.1 to obtain a set of sequents Π'. We clearly have $\models \Gamma \Rightarrow \Delta$ iff Π' is unsatisfiable. We now set up a refutation proof system where we *refute* Π' iff there is a refutation tree constructed from Π' as the root using the single schema of derivation rules:

$$\frac{A_1, \Gamma, \Pi \;\mid\; \cdots \;\mid\; A_m, \Gamma, \Pi}{(\Gamma \Rightarrow A_1, \ldots, A_m), \Gamma, \Pi}.$$

The rules reduce to axioms $\frac{}{(\Gamma \Rightarrow), \Gamma, \Pi}$ when $m = 0$. Using the ideas from the proof of Thm. 4.5.5, the system can be proved (refutationally) sound and complete.

Thus the refutation strategy is to try to satisfy the *guards* Γ in antecedents of sequents before splitting the refutation proof (if $m \geq 2$). This has direct consequences for the DPLL-style of SAT solving (see [NOT05, KV05]) where a good strategy might be not to make any decisions (free extensions in the terminology of [KV05]) until we have satisfied the guard of a sequent, and then use for the decision one of the positive atoms in the consequent of the sequent. If there is only one such ($m = 1$), then it is forced (see [KV05]) by unit propagation. In such a style of SAT solving, negative literals do not play any role except when they are forced by a conflict ($m = 0$).

The CA closure algorithm is an alternative to DPLL-style SAT solving modulo EUF as described above in Par. 4.7.2: The algorithm is driven by the congruence closure, and the propositional structure of formulas is secondary. Since we focus on assisted, not automated theorem proving, we have not yet conducted any experiments to compare real-world performance of the CA closure algorithm to state-of-the-art SAT-modulo-EUF solvers. We do not expect the CA closure algorithm to be competitive as presented here. However, by adapting ideas from SAT solving to CA closure, may lead to a practical tool for problems heavily dependent on equality. Lemma learning appears to be the most beneficial technique. It can be implemented by adapting the ideas from [NO03].

Since the CA closure algorithm is exponential in the worst case, it is not particularly suited for assisted theorem proving unless the depth of recursion in the

step 4.6.1(4) is limited. Limiting the depth to 0 yields the non-splitting algorithm of Par. 4.1.5 for Horn clauses, which we expect to be beneficial in a proof assistant. Feasibility of higher limits is subject to real-world testing.

5 Congruence Closure and Quantified Clauses

We have shown in the previous chapter that we can extend the congruence closure algorithm to decide validity of sequents of arbitrary quantifier-free sentences. In this chapter, we propose a method of deciding validity of a restricted class of sequents containing quantified sentences.

The experience with verification of programs in CL ([Kom09]) shows that many properties of programs can be expressed in the form of universally quantified clauses. This is also apparent in our example in Fig. 2.8. Many properties are proved by induction, and inductive hypotheses in proofs of clausal properties are also clausal. Moreover, CL programs are sets of universally quantified Horn clauses. Given this prominent role of generally quantified clauses, the automatic part of a proof assistant should therefore take them into account when deciding validity of sequents.

The current CL proof assistant is capable of using clausal definitions for simplification of assumptions and goals. The assistant in fact symbolically evaluates the definitions. Symbolic evaluation is possible due to constraints imposed on legal clausal definitions, and cannot be applied to general universally quantified clauses, not even when they are Horn clauses.

We propose an algorithm which derives logical consequences of a set Π consisting of quantifier-free sentences and universally quantified clauses restricted to a syntactic domain D (as defined in Par. 3.3.4). The domain D contains at least terms from the quantifier-free part of Π. Our intention is to let the user expand the domain with terms that she wants to use to instantiate clauses, and let the algorithm do the actual instantiation. Moreover, the domain can be also expanded automatically with terms obtained from symbolically evaluated clausal definitions, from simplification of basic arithmetic and other built-in operations, etc.

The easiest approach is to simply substitute all terms in D for all universally quantified variables and then use the CA closure from the previous chapter. Of course, this blind process produces too many useless clauses whose antecedents cannot be satisfied. We propose a more sophisticated method of finding instances of quantified clauses, which was outlined in Par. 1.3.2. This method uses equivalence classes maintained by the congruence closure algorithm to avoid producing instances with unsatisfiable antecedents. Moreover, whenever the algorithm produces an instance of a quantified clause, each term in the instance is provably equal to some term in D. The algorithm is sound, but necessarily incomplete. However, the algorithm is complete for so called domain-bounded clauses, which we will define in Par. 5.1.1.

We will start with the necessary definitions in Sect. 5.1, and give an overview of our algorithm in Sect. 5.2. Similarly to previous two chapters, we will define a formal system theoretically describing the working of our algorithm in Sect. 5.3. We will then outline the algorithm itself in Sect. 5.4. In Sect. 5.5, we will describe our initial

implementation and evaluation of the algorithm. We conclude with relationship to previous work in Sect. 5.6.

5.1 Domain-Bounded Theories

Unlike in previous two chapters, we will work with the standard semantics from Sect. 2.2 in this chapter. We will, however, still use the relation-based semantics from Sect. 3.3 for quantifier-free sentences. In order to capture the restriction of instances of clauses to a syntactic domain, we introduce below the concept of domain-bounded clauses.

5.1.1 Domain-bounded equalities, clauses, and theories. We will fix for this chapter a functional language $\mathcal{L}$, and a non-empty finite domain D of closed terms in $\mathcal{L}$ such that $\mathsf{o} \in D$. We will designate terms of $\mathcal{L}$ by a, b, c, d, and closed terms (usually from D) by s, t, u, v, possibly with subscripts.

For any term a, let us abbreviate the disjunction $(a = t_1 \vee a = t_2 \vee \cdots \vee a = t_n)$ where $\{\vec{t}\} = D$ to $\mathrm{D}(a)$. Let us further abbreviate the formula $(\mathrm{D}(a) \wedge a = b)$ to $a \approx b$, and call such formulas *D-bounded equalities*. We will write $\mathrm{D}(\vec{a})$ instead of $\mathrm{D}(a_1)$, $\ldots, \mathrm{D}(a_n)$; $\vec{a} \approx \vec{b}$ instead of $a_1 \approx b_1, \ldots, a_n \approx b_n$; and $\forall\vec{x}$ instead of $\forall x_1 \cdots \forall x_n$. If the particular variables are not important, we will denote sequences of universal quantifications (such as $\forall\vec{x}$, $\forall x \forall y \forall z$, etc.) by U.

A *D-bounded clause* is a sentence of the form $\forall\vec{x}(\vec{a} \approx \vec{b} \Rightarrow \vec{c} = \vec{d})$ such that (i) its consequent is non-empty, (ii) each variable of $\vec{x}$ occurs in the antecedent of the sequent, and (iii) each term of $\vec{c}$, $\vec{d}$ is either a closed term from D, or one of the variables $\vec{x}$.

Note that $\forall\vec{x}$ or $\vec{a} \approx \vec{b}$ may be empty sequences. In particular, atomic clauses $\Rightarrow s = t$ for terms $s, t \in D$ are D-bounded. Since the order of quantifiers $\forall\vec{x}$ does not affect the semantics of the D-bounded clause, and we will reorder them freely.

A theory T is *D-bounded* if every sentence of T is a D-bounded clause or an equality among the terms of D.

5.1.2 Restricting quantified clauses to a domain. Assume the automatic part of a proof assistant is started in order to prove a sequent $\Gamma \Rightarrow \Delta$. By Thm. 3.2.3 extended to quantified sentences, we can assume that the sequent is in a functional language.

Note that we could do the usual transformation of the sequent to a set of universally quantified clauses which is satisfiable iff the sequent is valid. This transformation consists of reduction to the negation normal form, prenexing, skolemization, and transformation to (equisatisfiable) CNF.

However, we do not consider this transformation appropriate in the context of assisted theorem proving, since it obscures the structure of the original sequent $\Gamma \Rightarrow \Delta$. We prefer extracting a subset $\Gamma' \Rightarrow \Delta'$ of the sequent which we can handle directly. This subset consists of possibly universally quantified clauses $\forall\vec{x}(\vec{a} = \vec{b} \Rightarrow \vec{c} = \vec{d})$ in

the antecedent Γ, and possibly existentially quantified formulas $\exists\vec{x}(\Rightarrow \vec{c} = \vec{d})$ and $\exists\vec{x}(c_1 = d_1 \wedge \cdots \wedge c_n = d_n)$ in the consequent Δ.

We then form a set of D-bounded clauses Π from $\Gamma' \Rightarrow \Delta'$ so that Π contains

1. $\forall\vec{x}\forall\vec{y}\forall\vec{z}(\vec{a} \approx \vec{b}, \vec{c} \approx \vec{y}, \vec{d} \approx \vec{z} \Rightarrow \vec{y} = \vec{z})$ for every $\forall\vec{x}(\vec{a} = \vec{b} \Rightarrow \vec{c} = \vec{d})$ of Γ' with a non-empty consequent, and some mutually distinct new variables $\vec{y}$ and $\vec{z}$;
2. $\forall\vec{x}(\vec{a} \approx \vec{b} \Rightarrow \mathtt{0} = \mathtt{1})$ for every $\forall\vec{x}(\vec{a} = \vec{b} \Rightarrow)$ of Γ';
3. $\forall\vec{x}(c_i \approx d_i \Rightarrow \mathtt{0} = \mathtt{1})$ for all $i = 1, \ldots, m$ for every $\exists\vec{x}(\Rightarrow \vec{c} = \vec{d})$ of Δ'.
4. $\forall\vec{x}(\vec{c} \approx \vec{d} \Rightarrow \mathtt{0} = \mathtt{1})$ for every $\exists\vec{x}(c_1 = d_1 \wedge \cdots \wedge c_n = d_n)$ of Δ'.

We will call Π the *D-restriction* of $\Gamma \Rightarrow \Delta$. While we show below Π is conservative over Δ, it is not an extension.

5.1.3 Theorem (Soundness of D-restriction). *Let $\Gamma \Rightarrow \Delta$ and Π be as in Par. 5.1.2. If $\Pi \vDash \mathtt{0} = \mathtt{1}$, then $\vDash \Gamma \Rightarrow \Delta$.*

Proof outline. Since $(\Gamma' \Rightarrow \Delta') \subseteq (\Gamma \Rightarrow \Delta)$, we will have $\vDash \Gamma \Rightarrow \Delta$ if $\vDash \Gamma' \Rightarrow \Delta'$. This is (similarly to Lemma 4.1.1) equivalent to $\Gamma', \{\, (A \Rightarrow) \mid A \in \Delta' \,\} \vDash (\Rightarrow)$, which is (similarly to Lemma 4.1.2) equivalent to $\Pi \vDash \mathtt{0} = \mathtt{1}$. □

5.2 Overview of the Algorithm for Domain-Bounded Clauses

Similarly to Sect. 3.1, we give a brief overview of the algorithm we aim at before going into detail.

As already noted in the introduction to this chapter, the basic idea of our decision algorithm is to instantiate D-bounded clauses so that there is a "reason to believe" that the antecedent of the instance can be satisfied. We use the equivalence classes maintained by the congruence closure algorithm to guide the instantiation.

The most interesting case is a clause of the form

$$\forall\vec{x}(f(\vec{a}) \approx g(\vec{b}), \Gamma \Rightarrow \Delta) \tag{1}$$

where $f(\vec{a})$ and $g(\vec{b})$ are not in the domain D. In order to produce instances of this clause with satisfied antecedents, we need to satisfy, among others, the D-bounded equality $f(\vec{a}) \approx g(\vec{b})$. We will therefore find such equivalence classes $[u] \subseteq D$ which contain some term $f(\vec{s})$ as well as some term $g(\vec{t})$. For each pair of such terms, if the antecedent of

$$\forall\vec{x}(\vec{a} \approx \vec{s}, \vec{b} \approx \vec{t}, \Gamma \Rightarrow \Delta) \tag{2}$$

is satisfied, then also the antecedent of (1) is satisfied.

Some of the equalities in the antecedent of (2) can be immediately eliminated. For instance, each $x_j \approx t$ can be eliminated by substituting t for each occurrence of x_j in (2); each $s \approx t$ can be eliminated if $[s] = [t]$. If iteration of these simplifications reduces the antecedent of (2) to the empty set, the (simplified) consequent of (2) will contain equalities among the terms of D. If there is only one equality, we have an atomic clause which can be added to the congruence closure. For more equalities in the simplified consequent of (2), we can split the proof as in the CA rule 4.5.1(3).

If the antecedent of (2) cannot be completely reduced, we can add simplified (2) to the set Π, and wait until consequences of other clauses change the equivalence so that the antecedent of (2) can be reduced. The clause (2) (and any its simplification by above eliminations) is simpler than (1), since the non-D terms $f(\vec{a})$ and $g(\vec{b})$ of (1) have been replaced with terms $\vec{a}$ and $\vec{b}$ of smaller depth. Therefore, production of new D-bounded clauses will eventually stop.

5.3 A Proof System For Domain-Bounded Theories

This section formalizes the instantiation process described in the previous section as a formal system extending the C and CA proof systems from preceding chapters. We will prove that the new formal system is complete for D-bounded theories.

5.3.1 The BC proof system for D-bounded theories. The following derivation rules for D-bounded theories formalize the reasoning sketched in the previous section. The order of quantifiers and the order of terms in D-bounded equalities are insignificant. We will call the formal system consisting of these rules as the *BC proof system* (for *b*ounded *c*lause).

$$\frac{U(\vec{a} \approx \vec{s}, \vec{b} \approx \vec{t}, \Gamma \Rightarrow \Delta), \Pi}{\Pi}$$
if $U(f(\vec{a}) \approx g(\vec{b}), \Gamma \Rightarrow \Delta) \in \Pi$, (1)
$f(\vec{a}), g(\vec{b}) \notin D$, $f(\vec{s}), g(\vec{t}) \in D$,
and $\Pi^{\mathrm{e}} \models_{\mathrm{e}} f(\vec{s}) = g(\vec{t})$;

$$\frac{U(\vec{a} \approx \vec{s}, \Gamma \Rightarrow \Delta), \Pi}{\Pi}$$
if $U(f(\vec{a}) \approx t, \Gamma \Rightarrow \Delta) \in \Pi$, $f(\vec{a}) \notin D$, (2)
$f(\vec{s}) \in D$, and $\Pi^{\mathrm{e}} \models_{\mathrm{e}} f(\vec{s}) = t$;

$$\frac{U(f(\vec{s}) \approx x, \vec{a} \approx \vec{s}, \Gamma \Rightarrow \Delta), \Pi}{\Pi}$$
if $U\forall x(f(\vec{a}) \approx x, \Gamma \Rightarrow \Delta) \in \Pi$, (3)
$f(\vec{a}) \notin D$, and $f(\vec{s}) \in D$;

$$\frac{U((\Gamma \Rightarrow \Delta)[x/t]), \Pi}{\Pi}$$
if $U\forall x(x \approx t, \Gamma \Rightarrow \Delta) \in \Pi$ and $t \in D$; (4)

$$\frac{U((\Gamma \Rightarrow \Delta)[x/t][y/t]), \Pi}{\Pi}$$
if $U\forall x\forall y(x \approx y, \Gamma \Rightarrow \Delta) \in \Pi$, (5)
and $t \in D$;

$$\frac{U(\Gamma \Rightarrow \Delta), \Pi}{\Pi}$$
if $U(s \approx t, \Gamma \Rightarrow \Delta) \in \Pi$, (6)
and $\Pi^{\mathrm{e}} \models_{\mathrm{e}} s = t$;

$$\frac{(\Rightarrow f(\vec{s}) = f(\vec{t})), \Pi}{\Pi}$$
if $f(\vec{s}), f(\vec{t}) \in D$, and $\Pi^{\mathrm{e}} \models_{\mathrm{e}} \vec{s} = \vec{t}$; (7)

$$\frac{(s = t), \Pi}{\Pi}$$
if $(\Rightarrow s = t) \in \Pi$; (8)

$$\frac{(\Rightarrow s_1 = t_1), \Pi \mid \cdots \mid (\Rightarrow s_m = t_m), \Pi}{\Pi}$$
if $(\Rightarrow \vec{s} = \vec{t}) \in \Pi$, and $m > 1$. (9)

If the antecedent of a clause $A \in \Pi$ is non-empty, A has the form $U(a \approx b, \Gamma \Rightarrow \vec{c} = \vec{d})$. Each of the terms a and b is either (i) a term from D, or (ii) a variable, or (iii) a possibly open term $f(\vec{a}) \notin D$ of the language $\mathcal{L}$. Clearly, every possible form of the D-bounded equality $a \approx b$ is covered up to symmetry by one of the rules (1–6).

Let T be a D-bounded theory. A *BC derivation for* T is a finite tree built by the derivation rules of the BC proof system from the root T. A D-bounded theory T *BC-proves* an equality $s = t$ for $s, t \in D$, in writing $T \vdash_{\mathrm{BC}} s = t$, if there is a BC derivation for T (called a *BC proof of* $s = t$ *from* T) such that $\Lambda^{\mathrm{e}} \models_{\mathrm{e}} s = t$ for every leaf Λ.

We wish to to prove Thm. 5.3.10, which asserts that the BC proof system is sound and complete for equalities among the terms of D. The theorem follows from several lemmas. As usual, soundness is simpler, and we prove it in Lemma 5.3.3. In order to prove completeness, we first define a well-founded ordering of D-bounded clauses (Par. 5.3.4, Lemma 5.3.5). The ordering is then used as a basis for inductive proofs. In Par. 5.3.6, we define weak BC saturation of D-bounded theories. We show existence of derivations leading to weakly saturated theories in Lemma 5.3.7. We construct a model of weakly saturated theories (Lemmas 5.3.8 and 5.3.9). The model is then used in an indirect proof of the completeness part of Thm. 5.3.10.

5.3.2 Lemma. *If the premise of any of the rules from Par. 5.3.1 is a D-bounded theory, each conclusion is also a D-bounded theory, which is a superset of the premise.* □

5.3.3 Lemma (Soundness of the BC proof system). *For every D-bounded theory T and any equality $s = t$ with $s, t \in D$, if $T \vdash_{\mathrm{BC}} s = t$, then $T \models s = t$.*

Proof. Take any equality $s = t$ with $s, t \in D$. We prove by complete induction on h that for any D-bounded theory Π, if there is a BC derivation of $s = t$ with the root Π of height h, then $\Pi \models s = t$. If the proof contains only the root, then $\Pi^{\mathrm{e}} \models_{\mathrm{e}} s = t$, hence $\Pi \models s = t$ by Cor. 3.3.9. Otherwise we assume the induction hypothesis for all conclusions (which are D-bounded by the previous lemma) of the bottom-most rule in the proof. We proceed by analyzing the possible bottom-most rules.

If the rule is 5.3.1(1), then there is $A \equiv U(f(\vec{a}) \approx g(\vec{b}), \Gamma \Rightarrow \Delta) \in \Pi$ with $f(\vec{a}), g(\vec{b}) \notin D$, there are $f(\vec{s}), g(\vec{t}) \in D$ such that $\Pi^{\mathrm{e}} \models_{\mathrm{e}} f(\vec{s}) = g(\vec{t})$, and the root node of the proof has one child labeled B, Π where $B \equiv U(\vec{a} \approx \vec{s}, \vec{b} \approx \vec{t}, \Gamma \Rightarrow \Delta)$. By the induction hypothesis $B, \Pi \models s = t$. We prove $\Pi \models s = t$ by showing that $\Pi \models B$. Take any structure $\mathcal{M}$ for the language $\mathcal{L}$ satisfying Π. Since B is universally quantified, we need to show that any expansion $\mathcal{M}'$ of $\mathcal{M}$ to $\mathcal{L}[\vec{\mathrm{x}}]$ satisfies $(\vec{a} \approx \vec{s}, \vec{b} \approx \vec{t}, \Gamma \Rightarrow \Delta)[\vec{x}/\vec{\mathrm{x}}]$. Assume $\mathcal{M}' \models (\vec{a} \approx \vec{s}, \vec{b} \approx \vec{t}, \Gamma)[\vec{x}/\vec{\mathrm{x}}]$. Due to congruence, we have $\mathcal{M}' \models (f(\vec{a}) = f(\vec{s}), g(\vec{b}) = g(\vec{t}))[\vec{x}/\vec{\mathrm{x}}]$. Moreover, since $\Pi^{\mathrm{e}} \models_{\mathrm{e}} f(\vec{s}) = g(\vec{t})$, we have $\Pi^{\mathrm{e}} \models f(\vec{s}) = g(\vec{t})$ by Cor. 3.3.9, thus necessarily $\mathcal{M}' \models f(\vec{s}) = g(\vec{t})$. By combining assumptions and above observations, we obtain

$\mathcal{M}' \vDash (f(\vec{a}) \approx g(\vec{b}), \Gamma)[\vec{x}/\vec{\mathrm{x}}]$, i.e., $\mathcal{M}'$ satisfies the antecedent of A. Since $\mathcal{M}'$ also satisfies A (since $\mathcal{M}$ satisfies Π), we have $\mathcal{M}' \vDash \Delta[\vec{x}/\vec{\mathrm{x}}]$.

A similar argument applies to rules 5.3.1(2–8). If 5.3.1(9) is the bottom-most rule, there is $(\Rightarrow \vec{s} = \vec{t}) \in \Pi$, and the proof has $m > 1$ branches, each starting with $(\Rightarrow s_j = t_j), \Pi$ for some $j \in \{1, \ldots, m\}$. Each branch is a BC proof of $s = t$, thus we have $(\Rightarrow s_j = t_j), \Pi \vDash s = t$ for every $j \in \{1, \ldots, m\}$ by IH. Since every model $\mathcal{M}$ of Π satisfies $(\Rightarrow \vec{s} = \vec{t})$, and thus it satisfies $s_j = t_j$ for some $j \in \{1, \ldots, m\}$, it satisfies also $s = t$ by the j-th instance of the induction hypothesis. □

5.3.4 An ordering of D-bounded clauses. The ordering defined in this paragraph will serve as a basis for showing completeness of the BC proof system and termination of its implementation. We say that a D-bounded clause P is a *parent* of a clause C (child), and write $P \succ_{\mathrm{P}} C$, iff $P \neq C$ and there is a set Σ of equalities among the terms of D such that one of the rules 5.3.1(1–6, 9) derives C, P, Σ from P, Σ. For example, the clause $U(f(\vec{a}) \approx g(\vec{b}), \Gamma \Rightarrow \Delta)$ is a parent of all clauses of the form $U(\vec{a} \approx \vec{s}, \vec{b} \approx \vec{t}, \Gamma \Rightarrow \Delta)$ such that $f(\vec{s}), g(\vec{t}) \in D$. We define the *ancestor* relation $\succ_{\mathrm{A}}$ among clauses as the transitive closure of $\succ_{\mathrm{P}}$. The following lemma shows two important properties of the ancestor relation. The necessary definitions and properties of multisets and orderings are summarized in Appendix A.

5.3.5 Lemma. *(i) The relation $\succ_{\mathrm{A}}$ defined in the above paragraph is a well-founded partial ordering of D-bounded clauses. (ii) The set $P_{\succ_{\mathrm{A}}}$ is finite for every clause P.*

Proof. We prove (i) by constructing a well-founded partial ordering of D-bounded clauses $\succ$, and showing that $\succ_{\mathrm{P}} \subseteq \succ$. Since $\succ_{\mathrm{A}}$ is a transitive closure of $\succ_{\mathrm{P}}$, it follows that also $\succ_{\mathrm{A}} \subseteq \succ$. Any infinite sequence descending in $\succ_{\mathrm{A}}$ would contradict well-foundedness of $\succ$, thus $\succ_{\mathrm{A}}$ must be well-founded (and also irreflexive).

The construction of $\succ$ requires a couple of auxiliary definitions: Let $\mathrm{d}_D(a)$ be the depth of the term a relative to D, defined by induction on the construction of terms: $\mathrm{d}_D(a) = 1$ if $a \in D$ or a is a variable; $\mathrm{d}_D(f(a_1, \ldots, a_n)) = 1 + \max\{\, \mathrm{d}_D(a_i) \mid 1 \leq i \leq n \,\}$ if $f(\vec{a}) \notin D$, with $\max\{\} = 0$. Clearly, the relation $s \succsim_D t$ iff $\mathrm{d}_D(s) \geq \mathrm{d}_D(t)$ is a quasi-ordering of terms and its strict part $\succ_D$ is well-founded. Let $\mathrm{M}(A)$ denote the multiset of all top-most occurrences of terms in a D-bounded clause A, more precisely: $\mathrm{M}(a = b) = \mathrm{M}(a \approx b) = \{\!\!\{a, b\}\!\!\}$; $\mathrm{M}(U(\Gamma \Rightarrow \Delta)) = \mathrm{M}(\Gamma \Rightarrow \Delta) = \biguplus_{A \in \Gamma, \Delta} \mathrm{M}(A)$. If A and B are clauses, we define $A \succsim B$ iff $\mathrm{M}(A) \succsim_D^{\mathrm{bag}} \mathrm{M}(B)$. Since $\succ_D$ is well-founded, so are $\succ_D^{\mathrm{bag}}$ and $\succ$.

Now we can prove $(\succ_{\mathrm{P}}) \subseteq (\succ)$. Take any P and C, and let us analyze the possible cases for P. If the antecedent of P is empty, then P is of the form $(\Rightarrow \vec{c} = \vec{d})$ for some n-tuples $\vec{c}, \vec{d}$ of terms. If n were 1, none of the rules 5.3.1(1–6, 9) could be applied to P to produce C. Therefore $n > 1$, and $C \equiv (\Rightarrow c_i = d_i)$ for some $i \in \{1, \ldots, n\}$ was obtained from P using the rule 5.3.1(9). From $\{\!\!\{\}\!\!\} \sim_D^{\mathrm{bag}} \{\!\!\{\}\!\!\}$ [A.2.2(i)], we can derive $\{\!\!\{c_i, d_i\}\!\!\} \sim_D^{\mathrm{bag}} \{\!\!\{c_i, d_i\}\!\!\}$ [twice A.2.2(ii)], hence $\mathrm{M}(P) = \{\!\!\{\vec{c}, \vec{d}\}\!\!\} \succ_D^{\mathrm{bag}} \{\!\!\{c_i, d_i\}\!\!\} = \mathrm{M}(C)$ [$2{\cdot}(n-1)$ times A.2.2(iii)]. This is equivalent to $P \succ C$.

In all other cases, P has a non-empty antecedent. The rules applicable to P and the form of C depend on D-bounded equalities occurring in the antecedent of P. If $P \equiv U(f(a_1,\ldots,a_m) \approx g(b_1,\ldots,b_n), \Gamma \Rightarrow \Delta)$ with $f(\vec{a}), g(\vec{b}) \notin D$, then the rule 5.3.1(1) is applicable. Hence C can be of the form $U(\vec{a} \approx \vec{s}, \vec{b} \approx \vec{t}, \Gamma \Rightarrow \Delta)$ for suitable $f(\vec{s}), g(\vec{t}) \in D$. Let M denote the multiset $\mathrm{M}(\Gamma \Rightarrow \Delta)$. Clearly $\mathrm{d}_D(f(\vec{a})) > \mathrm{d}_D(a_i)$ for all $1 \leq i \leq m$, and $\mathrm{d}_D(g(\vec{b})) > \mathrm{d}_D(b_i)$ for all $1 \leq i \leq n$. If f is not a nullary symbol, then $\mathrm{d}_D(f(\vec{a})) > 1 = \mathrm{d}_D(u)$ for all $u \in \{\vec{s}, \vec{t}\}$. The situation is similar for non-nullary g. Thus, if either f or g is non-nullary, we have $\mathrm{M}(P) = M \uplus \{\!|f(\vec{a}), g(\vec{b})|\!\} \succ_D^{\mathrm{bag}} M \uplus \{\!|\vec{a}, \vec{s}, \vec{b}, \vec{t}|\!\} = \mathrm{M}(C)$ by A.2.2(iii), hence $P \succ C$. If both f and g are nullary, then $\mathrm{M}(P) = M \uplus \{\!|f, g|\!\} \succ_D^{\mathrm{bag}} M \uplus \{\!|\,|\!\} = \mathrm{M}(C)$ by A.2.2(iii), so $P \succ C$, too.

The case for the rule 5.3.1(2) is similar, and 5.3.1(6) is almost trivial. The remaining cases (rules 5.3.1(3–5)) involve variables. Note that a term from D is always substituted for a variable, and so $\mathrm{M}(\Gamma \Rightarrow \Delta) \sim_D^{\mathrm{bag}} \mathrm{M}((\Gamma \Rightarrow \Delta)[x/t])$ for every $t \in D$ since $\mathrm{d}_D(x) = \mathrm{d}_D(t) = 1$. Hence, we obtain that the parent clause is greater in $\succ$ than the child clause by the properties A.2.2(ii, iii) of the multiset extension: In 5.3.1(4, 5), the parent contains more equalities than the child. In 5.3.1(3), we have $f(a_1,\ldots,a_m) \succ_D a_i$ and $f(\vec{a}) \succ_D s_i$ for all $1 \leq i \leq m$ as in the cases of 5.3.1(1, 2).

We prove (ii) by well-founded $\succ_{\mathrm{A}}$ induction. Take any clause P, and let us analyze its form as above. Assume first that the antecedent of P is empty. If P has a single equality in the consequent, it has no children, so $P_{\succ_{\mathrm{A}}}$ is empty. If there are $n > 1$ equalities in the consequent, the rule 5.3.1(9) can produce n children of P. They have no further descendants as each has a single equality in its consequent. Therefore, P has a finite number (n) of descendants.

Inspection of the rules 5.3.1(1–6) reveals that each rule can produce a finite number of children of any P with a non-empty antecedent: at most d^2, d, d, 1, d, and 1 respectively, where $d = |D|$. Since each child is smaller than P in the ordering $\succ_{\mathrm{A}}$, it has a finite number of descendants by the induction hypothesis. Therefore P has a finite number of descendants too. □

5.3.6 Saturation, canonical derivations. We define saturation of an atomic theory w.r.t. a derivation rule of the BC proof system as in Par. 3.4.3, and reuse the definition of weakly C-saturated theory. For a D-bounded theory Π, we denote by Π^{a} its *atomic part* $\Pi^{\mathrm{e}} \cup \{\, (\Rightarrow s = t) \mid (\Rightarrow s = t) \in \Pi \,\}$.

Similarly to Par. 4.5.2, we call a D-bounded theory Π *weakly BC-saturated* if it is saturated w.r.t. the derivation rules 5.3.1(1–6, 9), and Π^{a} is weakly C-saturated. A BC derivation is *canonical* if the atomic part of the premise of every application of any of the rules 5.3.1(1–6, 9) is a weakly C-saturated theory. We call a BC derivation *strictly monotonic* if every non-root node is a proper superset of its parent.

5.3.7 Lemma. *For every finite D-bounded theory T, there is a strictly monotonic canonical BC derivation for T with every leaf a weakly BC-saturated D-bounded theory.*

Proof. We will prove the lemma by induction which is based on the following auxiliary definitions: Let Γ_D denote the set $\bigcup_{s,t\in D}\{s = t, (\Rightarrow s = t)\}$, and for any D-bounded theory Π, let $\bar{\Pi}$ denote $\Pi \cup \bigcup_{A\in\Pi} A_{\succ_{\mathrm{A}}} \cup \Gamma_D$. Note that Lemma 5.3.5(ii) ensures that $\bar{\Pi}$ is finite whenever Π is. The set $\bar{\Pi}$ is such that every node in any BC derivation starting with Π is a subset of $\bar{\Pi}$. This can be easily proved by complete induction on the size of the derivation for any D-bounded theory Π.

Take any finite D-bounded theory T. The lemma is a direct consequence of the following proposition, which we will prove by induction on k:

For every $\Pi \subseteq \bar{T}$ such that $|\bar{T} \setminus \Pi| < k$, there is a derivation for Π with properties required by the lemma.

There is nothing to prove in the basis of the induction. Assume the proposition holds for k, and take any Π such that $|\bar{T} \setminus \Pi| < k+1$. If Π is weakly BC-saturated, then a single node derivation with the root Π trivially satisfies the requirements. Note that this is also the case of $\bar{T}$. If Π is not weakly BC-saturated, then at least one of the rules from Par. 5.3.1 can be applied to it so that each conclusion of the rule is a proper superset of Π. There is always only a finite number of such rule applications, and one can easily produce a simple computable function to choose one of them. In order to build a canonical derivation, the function must prefer the rules 5.3.1(7, 8).

The chosen rule produces $n \geq 1$ conclusions $\Pi_1, \ldots, \Pi_n$. Since we have considered only such rule application that each Π_i, $1 \leq i \leq n$, is a proper superset of Π, we also have $|\bar{T} \setminus \Pi_i| < k$. Thus, by the induction hypothesis, there exists a strictly monotonic canonical BC derivation for each Π_i with every leaf weakly BC-saturated. We construct a derivation with Π as the root, and the derivations for Π_i as its children. The derivation clearly satisfies the requirements. □

5.3.8 Lemma. *Let T be any weakly BC-saturated D-bounded theory, and let $\simeq$ be the equivalence over D induced by T^{e}. Let $\mathcal{M}$ be a structure for the language $\mathcal{L}$ with the first-order domain $M = (D/{\simeq}) \cup \{\emptyset\}$. For all $n \geq 0$, let each n-ary function symbol f of $\mathcal{L}$ be interpreted for all $\vec{x} \in M$ as*

$$f^{\mathcal{M}}(\vec{x}) = \begin{cases} [f(\vec{t})]_{\simeq} & \text{if } f(\vec{t}) \in D \text{ and } x_i = [t_i]_{\simeq} \text{ for all } i = 1, \ldots, n; \\ \emptyset & \text{otherwise.} \end{cases}$$

Then this definition is correct, and for all closed terms s, t, $f(\vec{t})$ of $\mathcal{L}$ we have:

(i) If $s \in D$, then $s^{\mathcal{M}} = [s]_{\simeq}$.

(ii) If $s, t \in D$, then $\mathcal{M} \vDash s = t$ iff $T^{\mathrm{e}} \vDash_{\mathrm{e}} s = t$.

(iii) $\mathcal{M} \vDash \mathrm{D}(f(\vec{t}))$ iff there is $f(\vec{s}) \in D$ such that $\mathcal{M} \vDash \vec{s} = \vec{t}$.

(iv) $\mathcal{M} \vDash \mathrm{D}(t)$ iff there $s \in D$ such that $t^{\mathcal{M}} = [s]_{\simeq}$.

(v) Let $x_1, \ldots, x_n$ be all variables in a term a, and let $\mathcal{M}'$ be an expansion of $\mathcal{M}$ to $\mathcal{L}[\mathsf{x}_1, \ldots, \mathsf{x}_n]$. If $\mathcal{M}' \vDash \mathrm{D}(a[\vec{x}/\vec{\mathsf{x}}])$, then $\mathsf{x}_i^{\mathcal{M}'} \neq \emptyset$ for all $1 \leq i \leq n$.

Proof. Note that the union defining M is disjoint since no equivalence class of $\simeq$ is empty. We first prove the correctness of the definition of $f^{\mathcal{M}}$: If for given $\vec{x} \in M$

there are terms $f(\vec{s}), f(\vec{t}) \in D$ such that $x_i = [s_i]_\simeq = [t_i]_\simeq$ for all $i = 1, \ldots, n$, then $s_i \simeq t_i$ for all $i = 1, \ldots, n$, hence $T^{\mathrm{e}} \models_{\mathrm{e}} \vec{s} = \vec{t}$. Since T is weakly saturated, $(\Rightarrow f(\vec{s}) = f(\vec{t})) \in T$ by rule 5.3.1(7). Thus $T^{\mathrm{e}} \models_{\mathrm{e}} f(\vec{s}) = f(\vec{t})$ by weak C saturation, whence $[f(\vec{s})]_\simeq = [f(\vec{t})]_\simeq$. The above definition of $f^{\mathcal{M}}$ is therefore correct.

The property (i) is easily shown by a simple induction on the structure of terms. Consequently $s^{\mathcal{M}} \neq \emptyset$ for $s \in D$ by the definition of M. As a corollary, by the definition of $\simeq$ we have for all $s, t \in D$, $T^{\mathrm{e}} \models_{\mathrm{e}} s = t$ iff $s^{\mathcal{M}} = [s]_\simeq = [t]_\simeq = t^{\mathcal{M}}$. The latter is equivalent to $\mathcal{M} \models s = t$. This proves (ii).

Obviously, the "if" ($\Leftarrow$) part of (iii) is due to the definition of $f^{\mathcal{M}}$. For the "only if" ($\Rightarrow$) part, take a closed term $f(\vec{t})$ of $\mathcal{L}$, and assume there is no $f(\vec{s}) \in D$ such that $\mathcal{M} \models \vec{s} = \vec{t}$, i.e., for every $f(\vec{s}) \in D$, there is i, $1 \leq i \leq n$, such that $s_i^{\mathcal{M}} = [s_i]_\simeq \neq t_i^{\mathcal{M}}$. Then $(f(\vec{t}))^{\mathcal{M}} = f^{\mathcal{M}}(t_1^{\mathcal{M}}, \ldots, t_n^{\mathcal{M}}) = \emptyset \notin D/\simeq$, i.e., $\mathcal{M} \not\models \mathrm{D}(f(\vec{t}))$.

The property (iv) is a simple consequence of (i) and of the meaning of the abbreviation $\mathrm{D}(t)$.

The property (v) follows from a slight variation of its contrapositive: If $\mathsf{x}_i^{\mathcal{M}'} = \emptyset$ for some i, $1 \leq i \leq n$, then $(a[\vec{x}/\vec{\mathsf{x}}])^{\mathcal{M}'} = \emptyset$. This is easily proved by induction on the structure of terms using (i). □

5.3.9 Lemma. *Let T be any weakly BC-saturated D-bounded theory. Then there is a structure $\mathcal{M}$ satisfying (i) $\mathcal{M} \models T$, and (ii) for all terms $s, t \in D$, $\mathcal{M} \models s = t$ iff $T^{\mathrm{e}} \models_{\mathrm{e}} s = t$.*

Proof. We will prove that the structure $\mathcal{M}$ from Lemma 5.3.8 possesses the required properties. We see that immediately for (ii) from 5.3.8(ii).

In order to show (i), notice first that any equality $(s = t) \in T$ is a member of T^{e}, and we have $\mathcal{M} \models s = t$ immediately from 5.3.8(ii). Other members of T are D-bounded clauses. We prove by well-founded $\succ_{\mathrm{A}}$ induction on A that $\mathcal{M} \models A$ for all D-bounded clauses $A \in T$.

Assume first that the antecedent of A is empty. Then $A \equiv (\Rightarrow \vec{c} = \vec{d})$ for some m-tuples $\vec{c}$ and $\vec{d}$ of terms. These terms must be in D; none can be a variable, since variables in the consequent of a D-bounded clause must also occur in its antecedent. For $m = 1$, we have $T^{\mathrm{e}} \models_{\mathrm{e}} c_1 = d_1$ from weak C saturation, and $\mathcal{M} \models c_1 = d_1$ from 5.3.8(ii). For $m > 1$, there is i, $1 \leq i \leq m$, such that $(\Rightarrow c_i = d_i) \in T$ due to BC saturation. Then $\mathcal{M} \models c_i = d_i$ as for $m = 1$, and consequently $\mathcal{M} \models (\Rightarrow \vec{c} = \vec{d})$.

If the antecedent of A in non-empty, A is of the form $\forall x_1 \cdots \forall x_n (a \approx b, \Gamma \Rightarrow \Delta)$. Take any expansion $\mathcal{M}'$ of $\mathcal{M}$ to $\mathcal{L}[\mathsf{x}_1, \ldots, \mathsf{x}_n]$ which satisfies the antecedent, i.e.,

$$\mathcal{M}' \models (a \approx b, \Gamma)[\vec{x}/\vec{\mathsf{x}}]. \tag{1}$$

We have to prove $\mathcal{M}' \models \Delta[\vec{x}/\vec{\mathsf{x}}]$. Let us assume without loss of generality that all variables $\vec{x}$ occur in the antecedent (otherwise denote by $\vec{x}$ those variables that do). If there were i, $1 \leq i \leq n$, such that $\mathsf{x}_i^{\mathcal{M}'} = \emptyset$, the D-bounded equality containing x_i would be false by 5.3.8(v), contradicting (1). Thus by 5.3.8(iv), there are terms $\vec{u} \in D$ such that $\mathsf{x}_i^{\mathcal{M}'} = [u_i]_\simeq$ for every $i = 1, \ldots, n$. Clearly,

$$\mathcal{M} \models (a \approx b, \Gamma)[\vec{x}/\vec{u}]. \tag{2}$$

That implies $\mathcal{M} \models \mathrm{D}(a[\vec{u}]), \mathrm{D}(b[\vec{u}])$. In order to prove $\mathcal{M}' \models \Delta[\vec{x}]$, we have to apply the case analysis discussed in Par. 5.3.1 to the equality $a[\vec{x}] \approx b[\vec{x}]$.

For example, assume that $a \equiv f(\vec{c})$ and $b \equiv g(\vec{d})$ for some f, g, $\vec{c}$, and $\vec{d}$. From 5.3.8(iii), we obtain terms $f(\vec{s}), g(\vec{t}) \in D$ such that

$$\mathcal{M} \models \vec{c}[\vec{x}/\vec{u}] = \vec{s}, \vec{d}[\vec{x}/\vec{u}] = \vec{t}. \tag{3}$$

By (2), $\mathcal{M} \models f(\vec{s}) = g(\vec{t})$, hence $T^{\mathrm{e}} \models_{\mathrm{e}} f(\vec{s}) = g(\vec{t})$ by 5.3.8(ii). Since T is weakly BC-saturated, the result $B \equiv \forall\vec{x}(\vec{c} \approx \vec{s}, \vec{d} \approx \vec{t}, \Gamma \Rightarrow \Delta)$ of application of the rule 5.3.1(1) is in T, thus $\mathcal{M} \models B$ by the induction hypothesis since $A \succ_{\mathrm{A}} B$. Consequently, $\mathcal{M}' \models (\vec{c} \approx \vec{s}, \vec{d} \approx \vec{t}, \Gamma \Rightarrow \Delta)[\vec{x}/\vec{\mathtt{x}}]$. Since $\mathcal{M}' \models \Gamma[\vec{x}/\vec{\mathtt{x}}]$ by (1), and $\mathcal{M}' \models (\vec{c} \approx \vec{s}, \vec{d} \approx \vec{t})[\vec{x}/\vec{\mathtt{x}}]$ follows from (3), we have $\mathcal{M}' \models \Delta[\vec{x}/\vec{\mathtt{x}}]$.

The proof is similar (and simpler) in the remaining cases. □

5.3.10 Theorem (Soundness and completeness of the BC proof system). *For every D-bounded theory T and any equality $s = t$ with $s, t \in D$, $T \vdash_{\mathrm{BC}} s = t$ iff $T \models s = t$.*

Proof. We have soundness ($\Rightarrow$) directly from Lemma 5.3.3. For completeness ($\Leftarrow$), we assume $T \nvdash s = t$ and show $T \not\models s = t$. Take a derivation tree with the root labeled by T and with weakly BC-saturated leaves, which exists according to Lemma 5.3.7. Since $T \nvdash s = t$, there is a leaf Π where $\Pi^{\mathrm{e}} \not\models_{\mathrm{e}} s = t$. Take a structure $\mathcal{M}$ for the theory Π whose existence is asserted by Lemma 5.3.9. Since Π is a superset of T, $\mathcal{M} \models T$ by 5.3.9(i), and at the same time $\mathcal{M} \not\models s = t$ by 5.3.9(ii), i.e., we have a model of T where $s = t$ does not hold, hence $T \not\models s = t$. □

5.4 A Congruence-Closure-Based Algorithm for Domain-Bounded Theories

We will now outline a decision algorithm which applies the rules of the BC proof system from Par. 5.3.1 in order to derive $\mathtt{o} = \mathtt{1}$ from a set of D-bounded clauses obtained as in Par. 5.1.2. We will call it the *BC closure algorithm*.

5.4.1 Triggering of clauses. Note first that we can treat antecedents of D-bounded clauses as sequences, rather than sets. Then, only one BC rule is applicable to each clause, and this rule is determined by the first equality in the clause's antecedent. It is possible to show that ordering of equalities in the antecedent into a sequence does not affect completeness of the system. Intuitively, since all equalities in the antecedent of a clause must hold to infer that the consequent holds, the particular order of equalities is irrelevant as long as all of them are considered. This observation is the first step towards a feasible algorithm for application of BC rules.

The second basic idea is to apply inference rules of the BC proof system when equivalence classes are joined by the congruence closure algorithm. We say that applications of inference rules are *triggered* by joins of equivalence classes. Since only one rule is applicable to each clause with a sequential antecedent, we will say

that clauses are triggered by joins. In the following three paragraphs, we will analyze which clauses are triggered when two classes are joined.

5.4.2 Triggering clauses of the form $U(f(\vec{a}) \approx g(\vec{b}), \Gamma \Rightarrow \Delta)$. Assume a set Π of D-bounded formulas contains a clause of the form $U(f(\vec{a}) \approx g(\vec{b}), \Gamma \Rightarrow \Delta)$ with neither $f(\vec{a})$ nor $g(\vec{b})$ from D. The only rule applicable to this clause is 5.3.1(1) and it can be applied when its conditions are met: when there are terms $f(\vec{s}), g(\vec{t}) \in D$ such that $\Pi^{\mathrm{e}} \models_{\mathrm{e}} f(\vec{s}) = g(\vec{t})$. The application of the rule leads to adding the clause $U(\vec{a} \approx \vec{s}, \vec{b} \approx \vec{t}, \Gamma \Rightarrow \Delta)$ to the set Π.

Now assume Π is saturated w.r.t. the rule 5.3.1(1), i.e., for the premise Π, every suitable clause C, and every suitable $f(\vec{s})$ and $g(\vec{t})$ from D, the conclusion of the rule is also Π. When we add a new equality $s = t$ to Π, we need to apply the rule possibly multiple times to restore saturation, forming a new saturated set Π'. In order to restore saturation efficiently, we need to find all triples $\langle U(f(\vec{a}) \approx g(\vec{b}), \Gamma \Rightarrow \Delta), \vec{s}, \vec{t}\rangle$ to which the rule 5.3.1(1) applies in $(s = t), \Pi$ but did not apply in Π alone. This set of new triples for a given clause $C \equiv U(f(\vec{a}) \approx g(\vec{b}), \Gamma \Rightarrow \Delta) \in \Pi$ is

$$
\begin{aligned}
&\{\, \langle C, \vec{s}, \vec{t}\rangle \mid f(\vec{s}), g(\vec{t}) \in D \text{ and } ((s = t), \Pi)^{\mathrm{e}} \models_{\mathrm{e}} f(\vec{s}) = g(\vec{t}) \,\} \\
&\qquad \setminus \{\, \langle C, \vec{s}, \vec{t}\rangle \mid f(\vec{s}), g(\vec{t}) \in D \text{ and } \Pi^{\mathrm{e}} \models_{\mathrm{e}} f(\vec{s}) = g(\vec{t}) \,\} \\
&\quad = \{\, \langle C, \vec{s}, \vec{t}\rangle \mid f(\vec{s}), g(\vec{t}) \in D \text{ and} \\
&\qquad\qquad (s = t), \Pi^{\mathrm{e}} \models_{\mathrm{e}} f(\vec{s}) = g(\vec{t}) \text{ and } \Pi^{\mathrm{e}} \not\models_{\mathrm{e}} f(\vec{s}) = g(\vec{t}) \,\} \\
&\quad = \{\, \langle C, \vec{s}, \vec{t}\rangle \mid f(\vec{s}), g(\vec{t}) \in D \text{ and} \\
&\qquad\qquad \Pi^{\mathrm{e}} \models_{\mathrm{e}} s = f(\vec{s}), t = g(\vec{t}) \text{ or } \Pi^{\mathrm{e}} \models_{\mathrm{e}} t = f(\vec{s}), s = g(\vec{t}) \,\} \\
&\quad = \{\, \langle C, \vec{s}, \vec{t}\rangle \mid f(\vec{s}), g(\vec{t}) \in D \text{ and } \Pi^{\mathrm{e}} \models_{\mathrm{e}} s = f(\vec{s}), t = g(\vec{t}) \,\} \qquad \text{(1a)} \\
&\qquad \cup \{\, \langle C, \vec{s}, \vec{t}\rangle \mid f(\vec{s}), g(\vec{t}) \in D \text{ and } \Pi^{\mathrm{e}} \models_{\mathrm{e}} t = f(\vec{s}), s = g(\vec{t}) \,\}. \qquad \text{(1b)}
\end{aligned}
$$

The last but one step is justified by Lemma 3.3.11. The rule 5.3.1(1) is applicable only to those clauses for which the above set of triples is non-empty. The set of such clauses is obviously

$$
\begin{aligned}
&\{\, U(f(\vec{a}) \approx g(\vec{b}), \Gamma \Rightarrow \Delta) \in \Pi \mid \\
&\qquad \Pi^{\mathrm{e}} \models_{\mathrm{e}} s = f(\vec{s}), t = g(\vec{t}) \text{ for some } f(\vec{s}), g(\vec{t}) \in D \,\} \\
&\quad \cup \{\, U(f(\vec{a}) \approx g(\vec{b}), \Gamma \Rightarrow \Delta) \in \Pi \mid \\
&\qquad\qquad \Pi^{\mathrm{e}} \models_{\mathrm{e}} t = f(\vec{s}), s = g(\vec{t}) \text{ for some } f(\vec{s}), g(\vec{t}) \in D \,\}.
\end{aligned} \qquad (2)
$$

If we define for each $u \in D$ and each $i = 1, 2$,

$$
\begin{aligned}
\mathit{clauses}_i^{(1)}[u] = \{\, &U(f_1(\vec{a}) \approx f_2(\vec{b}), \Gamma \Rightarrow \Delta) \in \Pi \mid \\
&\Pi^{\mathrm{e}} \models_{\mathrm{e}} u = f_i(\vec{s}) \text{ for some } f_i(\vec{s}) \in D \,\},
\end{aligned}
$$

then the set of clauses (2) can be expressed as

$$
(\mathit{clauses}_1^{(1)}[s] \cap \mathit{clauses}_2^{(1)}[t]) \cup (\mathit{clauses}_1^{(1)}[t] \cap \mathit{clauses}_2^{(1)}[s]). \qquad (3)
$$

Observe that if for some terms u, v there is a term $f(\vec{t}) \in D$ such that $\Pi^{\mathrm{e}} \models_{\mathrm{e}} u = f(\vec{t})$ and there is some $f(\vec{s}) \in D$ such that $\Pi^{\mathrm{e}} \models_{\mathrm{e}} v = f(\vec{s})$, then every clause $(f(\vec{a}) \approx g(\vec{b}), \Gamma \Rightarrow \Delta)$ belongs to both $\mathit{clauses}_1^{(1)}[u]$ and $\mathit{clauses}_2^{(1)}[v]$, even when Π^{e} does not imply $u = v$. Since the BC closure algorithm will keep the sets $\mathit{clauses}_i^{(1)}$ in data structures, we want to to avoid storing multiple times clauses common to different equivalence classes. A straightforward solution is to keep the sets of triggerable clauses not for each equivalence class but for each function symbol f, i.e., to have

$$\mathit{clauses}_1^{(1)}[f] = \{\, U(f(\vec{a}) \approx p, \Gamma \Rightarrow \Delta) \in \Pi \mid p \notin D \,\},$$
$$\mathit{clauses}_2^{(1)}[f] = \{\, U(p \approx f(\vec{a}), \Gamma \Rightarrow \Delta) \in \Pi \mid p \notin D \,\},$$

and for each $u \in D$ the set of function symbols

$$\mathit{fsyms}[u] = \{\, f \mid \Pi^{\mathrm{e}} \models_{\mathrm{e}} u = f(\vec{s}) \text{ for some } f(\vec{s}) \in D \,\}.$$

We now have for each $u \in D$ and $i = 1, 2$,

$$\mathit{clauses}_i^{(1)}[u] = \bigcup_{f \in \mathit{fsyms}[u]} \mathit{clauses}_i^{(1)}[f].$$

5.4.3 Triggering clauses of other forms. The analysis from the previous paragraph is also applicable to clauses of the form $U(f(\vec{a}) \approx r, \Gamma \Rightarrow \Delta)$ with $f(\vec{a}) \notin D$ and $r \in D$, and clauses of the form $U(r \approx u, \Gamma \Rightarrow \Delta)$ for r, $u \in D$. For a given clause $C \equiv U(f(\vec{a}) \approx t, \Gamma \Rightarrow \Delta)$, the set of pairs $\langle C, \vec{s}\rangle$ to which the rule 5.3.1(2) applies in $(s = t), \Pi$, but does not apply to in Π is

$$\begin{aligned}
&\{\, \langle C, \vec{s}\rangle \mid f(\vec{s}) \in D \text{ and } ((s = t), \Pi)^{\mathrm{e}} \models_{\mathrm{e}} f(\vec{s}) = r \,\} && \text{(1a)}\\
&\quad \smallsetminus \{\, \langle C, \vec{s}\rangle \mid f(\vec{s}) \in D \text{ and } \Pi^{\mathrm{e}} \models_{\mathrm{e}} f(\vec{s}) = r \,\}\\
&= \{\, \langle C, \vec{s}\rangle \mid f(\vec{s}) \in D \text{ and } (s = t), \Pi^{\mathrm{e}} \models_{\mathrm{e}} f(\vec{s}) = r \text{ and } \Pi^{\mathrm{e}} \not\models_{\mathrm{e}} f(\vec{s}) = r \,\}\\
&= \{\, \langle C, \vec{s}\rangle \mid f(\vec{s}) \in D \text{ and } \Pi^{\mathrm{e}} \models_{\mathrm{e}} s = f(\vec{s}), t = r \,\} && \text{(1b)}\\
&\quad \cup \{\, \langle C, \vec{s}\rangle \mid f(\vec{s}) \in D \text{ and } \Pi^{\mathrm{e}} \models_{\mathrm{e}} t = f(\vec{s}), s = r \,\}. && \text{(1c)}
\end{aligned}$$

As in the previous paragraph, the last step is justified by Lemma 3.3.11. The set of clauses to which the rule 5.3.1(2) is applicable for given s and t is the set of clauses for which the above set of pairs is non-empty, i.e.,

$$\begin{aligned}
&\{\, U(f(\vec{a}) \approx r, \Gamma \Rightarrow \Delta) \in \Pi \mid \Pi^{\mathrm{e}} \models_{\mathrm{e}} s = f(\vec{s}), t = r \text{ for some } f(\vec{s}) \in D \,\}\\
&\quad \cup \{\, U(f(\vec{a}) \approx r, \Gamma \Rightarrow \Delta) \in \Pi \mid \Pi^{\mathrm{e}} \models_{\mathrm{e}} t = f(\vec{s}), s = r \text{ for some } f(\vec{s}) \in D \,\}.
\end{aligned}$$

Analogously to the corresponding set 5.4.2(3) for the rule 5.3.1(1), the above set of clauses can be expressed as

$$(\mathit{clauses}_1^{(2)}[s] \cap \mathit{clauses}_2^{(2)}[t]) \cup (\mathit{clauses}_1^{(2)}[t] \cap \mathit{clauses}_2^{(2)}[s]) \qquad (2)$$

where

$$clauses_1^{(2)}[u] = \{\, U(f(\vec{a}) \approx r, \Gamma \Rightarrow \Delta) \in \Pi \mid \Pi^{\mathrm{e}} \models_{\mathrm{e}} u = f(\vec{s}) \text{ for some } f(\vec{s}) \in D \,\},$$
$$clauses_2^{(2)}[u] = \{\, U(f(\vec{a}) \approx r, \Gamma \Rightarrow \Delta) \in \Pi \mid \Pi^{\mathrm{e}} \models_{\mathrm{e}} u = r \,\}.$$

For u and v both being equal to some terms with a common top-most function symbol f, the problem of duplicated occurrences of clauses in $clauses_1^{(2)}[u]$ and $clauses_1^{(2)}[v]$ is similar as in the previous paragraph. The solution is analogous: We define

$$clauses_1^{(2)}[f] = \{\, U(f(\vec{a}) \approx r, \Gamma \Rightarrow \Delta) \in \Pi \mid r \in D \,\},$$

and have

$$clauses_1^{(2)}[u] = \bigcup_{f \in fsyms[u]} clauses_1^{(2)}[f].$$

Note that $clauses_2^{(2)}[u] \cap clauses_2^{(2)}[v] = \emptyset$ unless the terms u and v belong to the same equivalence class under Π^{e}.

Finally, the rule 5.3.1(6) for clauses of the form $U(r_1 \approx r_2, \Gamma \Rightarrow \Delta)$ has no parameters except for the clause. We thus only need to find the set of clauses to which 5.3.1(6) is applicable in $(s = t), \Pi$ and is not applicable in Π, which is

$$\{\, U(r_1 \approx r_2, \Gamma \Rightarrow \Delta) \in \Pi \mid \Pi^{\mathrm{e}} \models_{\mathrm{e}} s = r_1, t = r_2 \,\}$$
$$\cup \{\, U(r_1 \approx r_2, \Gamma \Rightarrow \Delta) \in \Pi \mid \Pi^{\mathrm{e}} \models_{\mathrm{e}} s = r_2, t = r_1 \,\}$$

by Lemma 3.3.11. Again, the set can be expressed analogously to 5.4.2(3) and (2) as

$$(clauses_1^{(6)}[s] \cap clauses_2^{(6)}[t]) \cup (clauses_1^{(6)}[t] \cap clauses_2^{(6)}[s]) \tag{3}$$

with

$$clauses_i^{(6)}[u] = \{\, U(r_1 \approx r_2, \Gamma \Rightarrow \Delta) \in \Pi \mid \Pi^{\mathrm{e}} \models_{\mathrm{e}} u = r_i \,\}, \quad i = 1, 2.$$

5.4.4 Non-triggered forms of clauses. In the above two paragraphs, we have not dealt with rules 5.3.1(3, 4, 5) and the forms of clauses $U(f(\vec{a}) \approx x, \Gamma \Rightarrow \Delta)$, $U(x \approx t, \Gamma \Rightarrow \Delta)$, and $U(x \approx y, \Gamma \Rightarrow \Delta)$ to which these rules apply. The reason is that these rules are not influenced by the equivalence induced by Π^{e}, and they can be applied as soon as a suitable clause is introduced into Π. Moreover, once one of the rules 5.3.1(3, 4, 5) has been applied all possible ways to a clause, the clause is no longer needed (except for determining saturation) and can be put aside.

5.4.5 The BC closure algorithm. The observations from the previous four paragraphs are the basis of the BC closure algorithm. The algorithm starts with a set of D-bounded clauses Π obtained as in Par. 5.1.2, and decides whether $\mathtt{0} = \mathtt{1}$ is derivable. It first initializes the congruence closure tree for the domain D and the atomic theory Π^{a}. The algorithm then continues as follows:

1. If $\Pi^{\mathrm{e}} \models_{\mathrm{e}} \mathtt{0} = \mathtt{1}$, then the algorithm terminates returning **true**.

2. A pending equality, i.e., some $(\Rightarrow s = t) \in \Pi$, is chosen. If there is no such equality, the algorithm continues with step 6.

3. The union of the sets of triggered clauses 5.4.2(3), 5.4.3(2, 3) is computed, and appropriate rules 5.3.1(1, 2, 6) are applied to the clauses in the union. New clauses produced by these rule applications are denoted by Ψ.

4. If Ψ is non-empty, the algorithm continues with the next step. Otherwise, the rules 5.3.1(3, 4, 5) are applied to the clauses in Ψ in all possible ways, yielding a set Ψ' of simplified clauses. This set is added to Π.

The rules $5.3.1(1, 2, 6)$ are applied to clauses in Ψ'. The set of new clauses produced by these applications is denoted by Ψ, and the step 4 is reiterated.

5. The equivalence classes of s and t are joined, i.e., the rule 5.3.1(8) is applied to $\Rightarrow s = t$, and the rule 5.3.1(7) is applied in all possible ways. The algorithm is then reiterated from step 1.

6. If there are no pending equalities, but there is some $(\Rightarrow \vec{s} = \vec{t}) \in \Pi$ with $n > 1$ equalities in Δ, the algorithm is recursively applied to $(\Rightarrow s_i = t_i), \Pi$ with a new copy of the congruence closure tree, for each $i = 1, \ldots, n$. The algorithm then terminates returning **true** iff all n recursive calls returned **true**. This corresponds to an application of the rule 5.3.1(9).

7. If the previous step does not apply, the algorithm terminates returning **false**.

5.4.6 Data structures. Since application of BC rules can produce a large number of clauses, it is important to store them in a way that facilitates fast retrieval of triggered clauses in the step 3 of the BC closure algorithm. Whenever we add an equality $s = t$ to Π, we need to compute the union of intersections

$$(\mathit{clauses}_1[s] \cap \mathit{clauses}_2[t]) \cup (\mathit{clauses}_1[t] \cap \mathit{clauses}_2[s]) \tag{1}$$

where

$$\mathit{clauses}_i[u] = \mathit{clauses}_i^{(1)}[u] \cup \mathit{clauses}_i^{(2)}[u] \cup \mathit{clauses}_i^{(6)}[u], \quad i = 1, 2.$$

Note that if k, $l \in \{1, 2, 6\}$ are such that $k \neq l$, then the sets $\mathit{clauses}_i^{(k)}[u]$ and $\mathit{clauses}_j^{(l)}[v]$ are disjoint for all terms u and $v \in D$ and all $i, j \in \{1, 2\}$. The union (1) is therefore equal to the union of 5.4.2(3), 5.4.3(2, 3).

In order to compute the union (1) efficiently, the BC algorithm needs to store the sets

$$\begin{aligned} &\mathit{clauses}'_1[f] = \mathit{clauses}_1^{(1)}[f] \cup \mathit{clauses}_1^{(2)}[f] && \mathit{clauses}'_1[u] = \mathit{clauses}_1^{(6)}[u] \\ &\mathit{clauses}'_2[f] = \mathit{clauses}_2^{(1)}[f] && \mathit{clauses}'_2[u] = \mathit{clauses}_2^{(2)}[u] \cup \mathit{clauses}_2^{(6)}[u] \end{aligned}$$

and the sets $\mathit{fsyms}[u]$. Moreover, the sets

$$\mathit{args}[u, f] = \{\, (\vec{t}) \mid f(\vec{t}) \in D \text{ and } \Pi^{\mathrm{e}} \vDash_{\mathrm{e}} u = f(\vec{t}) \,\}$$

are needed to efficiently compute clauses resulting from the rules 5.3.1(1, 2).

After the set of triggered clauses (1) is computed and rules applied, new clauses must be added to their respective sets, and the sets of clauses $clauses'_i[s]$ and $clauses'_i[t]$ must be merged, as well as the sets of function symbols $fsyms[s]$ and $fsyms[t]$, and the sets of arguments $args[s, f]$ and $args[t, f]$ for all function symbols f.

With suitable indexing and encoding, all these sets can be represented in trees similar to equivalence trees, and then the procedure Join-Classes can be applied to these trees to merge the sets of clauses, function symbols, and arguments. Alternatively, a data structure supporting set operations efficiently, such as the Patricia-based trees of [OG98], can be used to directly represent the above sets.

5.4.7 Termination, correctness, and complexity. Since the BC closure algorithm produces new clauses by application of the rules of the BC proof system, soundness of the algorithm follows from soundness of the proof system. Completeness (relative to completeness of the BC proof system) follows from the note in Par. 5.4.1. Termination of the algorithm follows from Lemma 5.3.5.

Complexity of the algorithm is bounded by the number of clauses which can be derived from the initial set Π. This number can be roughly estimated as follows: Application of a rule to a clause can produce up to $|D|^2$ child clauses (cf. the proof of Lemma 5.3.5). Depth of terms relative to D in the child clauses is lower compared to the parent clause. So, about $O(|D|^d)$ clauses can be produced, where d is the maximum of depths of terms (relative to D) occuring in Π.

5.5 Implementation and Evaluation

5.5.1 Implementation of the BC closure algorithm. We have implemented the BC closure algorithm for evaluation purposes in the functional programming language Haskell. The implementation slightly differs from the algorithm described in the previous section.

The first difference is that the implemented version allows open terms in consequents of clauses. When a child clause with empty antecedent is derived, then its consequent possibly contains equalities among new closed terms which are not in the domain. Such equalities are saved for later processing. If $\mathtt{0} = \mathtt{1}$ cannot be derived in the current domain, the domain is expanded with new terms from the saved equalities, the saved equalities are added to the congruence closure, and the algorithm is reiterated. A fixed number of such iterations is allowed (we have used 3 during testing).

The second difference is that the implementation does not split the proof, i.e., only clauses with one equality in the consequent are allowed. This restriction mirrors the current CL proof assistant, which does not automatically split either.

5.5.2 Evaluation. The implementation was tested on a collection of solved exercises from the course Specification and Verification of Programs. The collection was created and provided to us by our colleague Ján Komara. The exercises contain

formal proofs of properties of various CL programs operating on lists and tree-like data structures (binary trees, binary search trees, arithmetic expressions, and propositional formulas). These programs are described in Komara's thesis [Kom09, Chaps. 6, 7].

In the current CL proof assistant, the automatic part is run after each user command. We have modified the proof assistant to output sequents describing the state of the proof at these points. We have run our implementation of the BC closure algorithm on these sequents extended with a set of basic axioms of arithmetic and the pairing function and list concatenation.

Our implementation was able to fully prove 243 (53 %) of the total of 457 theorems. In 73 theorems (16 %), our implementation was unable to prove even a single sequent provable by the current proof assistant. These results show that the BC closure algorithm is not as powerful as the canonization-based automatic part of the current proof assistant. However, these results were expected. Reasoning in arithmetic using axioms requires introduction of many new terms to take advantage of associativity and commutativity of arithmetic operations. In order to match the power and performance of the current CL proof assistant, we will have to combine the BC closure with decision procedures for theories of linear arithmetic and pairing in the Nelson–Oppen style (cf. Par. 4.7.2).

The aim of the BC closure algorithm, however, is not to decide properties of built-in operations. Its main purpose is to enable automatic use of user-proved lemmas and of quantified assumptions, such as more complex induction hypotheses. In the specification and verification exercises, there were 1180 sequents which the current automatic part could not prove, and were followed by a manual instantiation of lemmas or quantified assumptions. The BC algorithm was able to prove 443 sequents (37.5 %) preceding a manual instantiation. We consider this to be a promising result. The number of proved sequents should be further improved by combination with decision procedures for theories of built-in operations, which should make better use of the instances produced by the BC closure.

As mentioned above, our implementation expands the domain in 3 iterations. Increasing the number of iterations did not substantially improve the number of proved theorems, but caused an excessive increase of the running time.

5.6 Related Work

5.6.1 Simplify. After we have designed the BC proof system and the BC closure algorithm, we came across the description [DNS05] of the theorem prover Simplify, mentioned already in Par. 4.7.2. Simplify is a fully automatic prover aimed at verification of safety properties of imperative programs, such as access to arrays within bounds, handling of all exceptions, correct use of locks. The core of Simplify is an SMT solver (cf. Par. 4.7.2) with decision procedures for the theories of equality, linear rational arithmetic, partial orders, and array access and updates. Simplify can also

handle quantified formulas using an incomplete, but practically effective technique of heuristic pattern-matching of terms.

The congruence closure and combination of decision procedures in Simplify represent terms as an oriented graph, called the *E-graph*. From each universally quantified formula, a set (called a *trigger*) of terms with variables (*patterns*) is selected so that all quantified variables of the formula occur in the trigger. A pattern t matches a term s represented in the E-graph if there is a substitution θ such that $t\theta = s$ holds under the equivalence represented in the E-graph. Whenever there is a common substitution for which each pattern of a trigger matches a term represented in the E-graph, the quantified formula is instantiated using the substitution. Finding such a substitution is an NP-complete problem (see [Koz77] where it is called the schema validity problem, or [Fon04, pp. 151–153] where a more modern name *non-simultaneous rigid E-unification* is used).

For example, in the formula $\forall x\big(f(g(x)) = f(x)\big)$, the set of terms $\{f(g(x))\}$ can be selected as a trigger. If the current partial assignment contains a term $f(a)$, and the equality $g(b) = a$, then the E-graph represents also the term $f(g(b))$, which is matched by the term $f(g(x))$. The prover instantiates the formula with $x := b$, possibly introducing into the E-graph a new term $f(b)$ and the equality $f(g(b)) = f(b)$.

Since matching can introduce new terms, it is prone to matching loops: terms introduced by matching can be matched producing more new terms. Simplify provides only heuristics to stop matching loops. The SMT solving part of Simplify is organized according to the non-exhaustive eager schema (see Par. 4.7.2) where the decision procedures for theories are incrementally informed of extensions to partial truth assignments. Matching is also performed incrementally, and a number of optimizations is necessary to eliminate repeated matching in successive steps of construction of a partial truth assignment.

Simplify's heuristic matching is also used in theorem provers based on SMT solving with lazy proof explication Verifun ([FJOS03, FJS04]), and Zap ([LMO05, BLM05]). The provers implement different strategies of integration of matching and SMT solving: Simplify and Verifun add instantiated formulas produced by matching to the set of clauses processed by the SMT solver. Zap uses a two-tier technique of handling quantifiers: The main SAT solver reasons about the propositional structure of the input set of formulas treating quantified formulas as atoms. Once it finds a satisfying assignment, theory decision procedures and matching are used together with another, "little" SAT solver in an attempt to refute the assignment. If it is found unsatisfiable, theory decision procedures or the little solver explicate the proof to the main solver. This prevents overwhelming the main solver with instances of quantified formulas produced by matching.

5.6.2 Comparison of Simplify and the BC closure algorithm. Simplify and our BC closure algorithm share the idea of matching open terms against the congruence closure data structure (E-graph or equivalence tree respectively). In Simplify, matching is explicit. Our algorithm performs matching indirectly and incrementally

(first on the top level of terms, then on the second level, etc.) by application of rules of the BC proof system. Simplify's pattern-element optimization is similar to our triggering of clauses by equivalence class joins.

There are, however, notable differences between matching in Simplify and in our algorithm.

1. Simplify's matching is unconstrained by equalities of top-level terms in antecedents of quantified clauses. For a quantified clause

$$C \equiv \forall x\big(f(g(x)) = h(x) \Rightarrow g(x) = x\big),$$

Simplify will select either $\{f(g(x))\}$ or $\{h(x)\}$ as a trigger. Suppose $\{f(g(x))\}$ is selected. Simplify produces an instance of the above clause whenever a closed term $f(g(t))$ occurs in the E-graph. This may lead to introduction of a new term $h(t)$.

The BC closure algorithm takes a much more conservative approach. The algorithm will derive an instance of C if (i) there is $f(g(t))$ in the equivalence tree, (ii) $f(g(t))$ is equal to some $h(s)$, and (iii) t is equal to s.

In other words, Simplify matches *selected terms* in quantified clauses against the E-graph, while the BC closure algorithm matches *all equalities* in quantified clauses against the equivalence tree.

2. There is no limit on new terms introduced by matching in Simplify, but the BC closure algorithm is strictly constrained to a fixed domain. As noted in Sect. 5.5, our actual implementation expands the domain, but only a fixed number of iterations is allowed, and increasing the number did not significantly improve the results of the BC algorithm.

3. The authors of Simplify described matching operationally as a heuristic. We have characterized the BC closure algorithm logically.

6 Conclusion and Future Work

We have presented the theoretical background for two changes in the current design of the programming and proof system CL. These changes will be incorporated into the upcoming reimplementation of CL.

(i) We have proposed to replace the underlying first-order theory of Peano Arithmetic with a weak second-order arithmetic theory CL_2. CL_2 is equivalent to RCA_0, but it is better suited as the basis for a programming language. A basic, non-modular programming language can be bootstrapped in CL_2 just like in PA. In addition, CL_2 will enable an elegant logical explanation of modular programming with encapsulation.

The change of the underlying theory will be accompanied by a change of the formal system for verification of programs. The new formal system with one rule, the valuation tree calculus, will simplify the internal structure of formal proofs in the new implementation of CL. The calculus permits to present theory extensions as proofs. This removes the need for a separate theory-extension mode present in the current version of CL. Removal of this mode will further simplify the new implementation.

(ii) We have designed the BC closure algorithm for the automatic part of the proof assistant. The algorithm builds on the congruence closure algorithm, and enables automatic use of lemmas in universally quantified clausal form. Usefulness of the BC closure algorithm is indicated by our practical evaluation, and by existence of a similar practically applied algorithm.

We have also given a new presentation of the congruence closure algorithm as a transformation of set-theoretical equivalence trees, and proposed an algorithm for decision of satisfiability of variable-free clauses with equality using the notion of anticongruence.

In the near future, we will with our colleagues commence the work on the actual reimplementation of CL. Implementation of the modular features enabled by the move to the second-order logic will be challenging especially from the user interaction point of view. We will have to retain the current version's highly interactive approach to program and proof development.

The implementation of the automatic part of the proof assistant will be based on the preliminary implementation of the BC closure algorithm. However, substantial changes will be required to

(i) support explanation of proofs outlined in Par. 1.3.2 in the introduction; and

(ii) integrate the BC closure algorithm with decision procedures for the built-in CL operations in the Nelson–Oppen style, to achieve power and performance parity with the current implementation.

In the long term, we plan to investigate further which programming phenomena can be explicated in second-order arithmetic. We would especially like to explain various imperative features such as input/output operations, and in-place memory updates, which we have started to investigate earlier ([Kľu03, Kľu04]). Imperative features have been incorporated into the functional programming language Haskell without compromising its declarative nature by using the algebraic category of monads. However, monads require higher-order logic, and we would like to stay in the realm of weak second-order arithmetic. Another interesting area of future research is program transformation, especially oriented towards building complex verified programs from small verified components.

A Elements of The Theory of Orderings

The purpose of this short chapter is to introduce a small part of the theory of orderings used in term rewriting. We introduce the common concepts of well-founded ordering, and the multiset (bag) extension of orderings as found, e.g., in [DP01]. We also define our own extension of well-founded orderings to finite sets.

Orderings are a common tool for proving termination of term-rewriting systems. They are also used to guide theorem-proving algorithms, mainly those based on resolution. We need in Sect. 5.3 only a few basic definitions and theorems.

A.1 Orderings and Well-Foundedness

A.1.1 Kinds of orderings, well-foundedness. A reflexive and transitive relation over a set S is called a *quasi-ordering* of S (sometimes also pre-ordering). Given a quasi-ordering $\succsim$, let $x \sim y$ denote that $x \succsim y$ and $y \succsim x$, and let $x \succ y$ denote that $x \succsim y$ but not $y \succsim x$. The relation $\succ$ is called the *strict part* of $\succsim$, and it is a *strict partial ordering* (an irreflexive and transitive relation). A strict partial ordering of a set S is *well-founded* if there is no infinite descending sequence $x_1 \succ x_2 \succ \cdots \succ x_n \succ \cdots$ of elements x_i of S. We write $x_{\succ}$ to denote the set $\{\, y \mid x \succ y \,\}$ of elements smaller than x in a strict partial ordering $\succ$.

The set-theoretic theorem A.1.3 validates the use of well-founded induction in some proofs in the later chapters of this thesis.

A.1.2 Lemma. *Let $\succ$ be a transitive relation, and let $\succ'$ be a strict partial ordering. If $\succ \subseteq \succ'$, then $\succ$ is a strict partial ordering. Moreover, if $\succ'$ is well founded, then so is $\succ$.*

Proof. Existence of x such that $x \succ x$ would contradict irreflexivity of $\succ'$. Similarly, existence of an infinite sequence descending in $\succ'$ would contradict well-foundedness of $\succ'$. □

A.1.3 Theorem (The principle of well-founded induction). *Let $\succ$ be a well-founded quasi-ordering of the set S, and let P be a property (a set). If every element x of S has the property P (i.e., $x \in P$) whenever every $y \in S$ such that $x \succ y$ has the property P, then $S \subseteq P$.* □

A.2 The Multiset Extension

A.2.1 Multisets. A *multiset* (or bag) M on a set S is a map from S to natural numbers. Informally, it is a collection of elements of S in which each element can occur more than once. We will use the notation $\wr t_1, \ldots, t_n \int$ for enumerated multisets. For multisets M_1 and M_2 on S, we define their *union* as the multiset $M_1 \uplus M_2$ satisfying $(M_1 \uplus M_2)(x) = M_1(x) + M_2(x)$ for all $x \in S$, and their *intersection* as the multiset $M_1 \pitchfork M_2$ satisfying $(M_1 \pitchfork M_2)(x) = \min\{M_1(x), M_2(x)\}$ for all $x \in S$. A multiset M is *finite* if the set $\{\, x \in S \mid M(x) > 0 \,\}$ is finite.

A.2.2 Multiset extension of quasi-orderings. The *multiset extension* of a quasi-ordering $\succsim$ of a set S is an ordering $\succsim^{\text{bag}}$ of finite multisets on S with following properties:

(i) $\wr \int \sim^{\text{bag}} \wr \int$;
(ii) $M \uplus \wr s \int \sim^{\text{bag}} N \uplus \wr t \int$ if $s \sim t$ and $M \sim^{\text{bag}} N$;
(iii) $M \uplus \wr s \int \succ^{\text{bag}} N \uplus \wr t_1, \ldots, t_n \int$ if $n \geq 0$, $s \succ t_i$ for all $1 \leq i \leq n$, and $M \succsim^{\text{bag}} N$.

A.2.3 Theorem ([DM79]). *If $\succeq$ is a quasi-ordering of a set S, then its multiset extension $\succeq^{\text{bag}}$ is a well defined quasi-ordering of multisets on S. If $\succ$ is well-founded, then so is $\succ^{\text{bag}}$.* □

Bibliography

[Apt88] Krzysztof Apt. *Introduction to Logic Programming.* Technical Report CS-TR-87-35. University of Texas in Austin, Austin, Texas, 1988.

[ARR03] Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183:140–164, Elsevier, 2003.

[Avi02] Jeremy Avigad. Saturated models of universal theories. *Annals of Pure and Applied Logic*, 118(3):219–234, Elsevier, 2002.

[BCD+98] Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. *ELAN V 3.3 User Manual.* INRIA Lorraine & LORIA, Nancy, France, third ed., December 1998.

[BDS02] Clark W. Barrett, David L. Dill, and Aaron Stump. A generalization of Shostak's method for combining decision procedures. In A. Armando, ed.: *Frontiers of Combining Systems*, LNCS 2309, pp. 132–146. Springer, 2002.

[BG01] Leo Bachmair and Harald Ganzinger. Resolution Theorem Proving. In A. Robinson and A. Voronkov, eds.: *Handbook of Automated Reasoning*, vol. I, ch. 7, pp. 371–443. Elsevier Science, 2001.

[BLM05] Thomas Ball, Shuvendu K. Lahiri, and Madanlal Musuvathi. *Zap: Automated Theorem Proving for Software Analysis.* Technical Report MSR-TR-2005-137, Microsoft Research, Redmond, WA, 2005.

[Bun99] Alan Bundy. *A Survey of Automated Deduction.* Informatics Research Report No. 1. Institute of Representation and Reasoning, Division of Informatics, University of Edinburgh, April 1999.

[Bus95] Samuel R. Buss. On Herbrand's Theorem. In *Logic and Computational Complexity*, LNCS 960, pp. 195–209, Springer, 1995.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, Walter Rance Cleaveland, J. F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Developement System.* Prentice-Hall, NJ, 1986.

[CDT06] The Coq Development Team. *The Coq Proof Assistant Reference Manual. Version 8.0.* INRIA, 2006.

[CIMP03] David Carlisle, Patrick Ion, Robert Miner, and Nico Poppelier. *Mathematical Markup Language (MathML) Version 2.0 (Second Edition).* The World Wide Web Consortium (MIT, ERCIM, Keio), 2003. Available at http://www.w3.org/TR/2003/REC-MathML2-20031021/.

[CL] *CL: Programming Language and Proof Assistant.* Available at http://ii.fmph.uniba.sk/cl/.

[DGLP04] Heidi E. Dixon, Matthew L. Ginsberg, Eugene M. Luks, and Andrew J. Parkes. Generalizing Boolean Satisfiability II: Theory. *Journal of Artificial Intelligence Research*, 22:481–534, 2004.

[DGP04] Heidi E. Dixon, Matthew L. Ginsberg, and Andrew J. Parkes. Generalizing Boolean Satisfiability I: Background and Survey of Existing Work. *Journal of Artificial Intelligence Research*, 21:193–243, 2004.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DM79] Nachum Dershowitz and Zohar Manna. Proving Termination with Multiset Orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM*, 52(3):365–473, 2005. Also Technical Report HPL-2003-148, HP Labs, Palo Alto, CA, 2003. Available at http://www.hpl.hp.com/techreports/2003/HPL-2003-148.ps

[DP01] Nachum Dershowitz and David A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, eds.: *Handbook of Automated Reasoning*, vol. I, ch. 9, pp. 535–610. Elsevier Science, 2001.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[DST80] Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the Common Subexpression Problem. *Journal of the ACM*, 27(4):758–771, 1980.

[ES03] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6^{th} International*

Conference, SAT 2003, Selected Revised Papers, LNCS 2919, pp. 502–518, Springer, 2003.

[Fer05] Fernando Ferreira. A Simple Proof of Parsons' Theorem. *Notre Dame Journal of Formal Logic*, 46(1):83–91, University of Notre Dame, 2005.

[FJOS03] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem Proving using Lazy Proof Explication. In *Proc. 15th International Conference on Computer-Aided Verification, July 2003*, LNCS 2725, pp. 355–367, Springer, 2003.

[FJS04] Cormac Flanagan, Rajeev Joshi, and James B. Saxe. *An Explicating Theorem Prover for Quantified Formulas.* Technical Report HPL-2004-199, HP Labs, Palo Alto, CA, 2004.

[Fon04] Pascal Fontaine. *Techniques for Verification of Concurrent Systems with Invariants.* PhD thesis, Institut Montefiore, Université de Liège, Belgium, 2004.

[GHN04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast Decision Procedures. In *16th International Conference on Computer Aided Verification (CAV), Boston, July 2004*, LNCS 3114, pp. 175–188, Springer, 2004.

[HHL10] *Haskell Hierarchical Libraries.* http://www.haskell.org/ghc/docs/latest/html/libraries/. Retrieved April 23, 2010.

[HP93] Petr Hájek and Pavel Pudlák. *Metamathematics of First-Order Arithmetic.* Perspectives in Mathematical Logic, Volume 3, Springer, 1993.

[HP06] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16:2 (2006), pp. 197–217. Available at http://www.soi.city.ac.uk/~ross/papers/FingerTree.html

[Kľu01] Ján Kluka. *Specification, implementation, and verification of sequences in Peano Arithmetic.* Diploma thesis. Comenius University, Bratislava, 2001.

[Kľu03] Ján Kluka. A Simple Semantics for Destructive Updates. In B. ten Cate: *Proc. 15th ESSLLI Student Session*, Kurt Gödel Society and Technical University in Vienna, Vienna, August 2003.

[Kľu04] Ján Kluka. *Prepisovanie na mieste v deklaratívnom programovaní.* (In-Place Updates in Declarative Programming. In Slovak only.)

Written part of the dissertation examination. Comenius University, Bratislava, January 2004.

[Kľu05] Ján Kľuka. Congruence-Anticongruence Closure. In M. Češka, J. Gruska, A. Kučera, eds.: *Proc. 1st Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2005)*, FI MU Report Series, FIMU-RS-2005-11. Masaryk University, Brno, September 2005.

[Kľu10] Ján Kľuka. *Completeness of Second-Order Arithmetic: A Case Study in Modular Theorem Proving.* Available at http://dai.fmph.uniba.sk/~kluka/modular-completeness.pdf.

[Kom09] Ján Komara. *Špecifikácia a verifikácia programov v Peanovej aritmetike. (Specification and Verification of Programs in Peano Arithmetic.)* Dissertaion. Comenius University, Bratislava, 2009.

[Koz77] Dexter Kozen. Complexity of Finitely Presented Algebras. In *Proc. of the Ninth Annual ACM Symposium on Theory of Computing*, 164–177, ACM, New York, 1977.

[Kre02] Christoph Kreitz. *The Nuprl Proof Developement System, Version 5. Reference Manual and User's Guide.* Unpublished manuscript, Cornell University, Ithaca, NY, 2002. Available at http://www.cs.cornell.edu/Info/Projects/NuPRL/html/nuprl5docs.html.

[KV05] Ján Kľuka and Paul J. Voda. *Forcing in DPLL Satisfiability Solving.* Technical report. Available at http://dai.fmph.uniba.sk/~voda/.

[KV09] Ján Kľuka and Paul J. Voda. *A Simple and Practical Valuation Tree Calculus for First-Order Logic.* Extended abstract. Talk presented at Computability in Europe 2009, Heidelberg, Germany, July 2009. Available at http://dai.fmph.uniba.sk/~voda/ext-proof.pdf

[LMO05] K. Rustan M. Leino, Madan Musuvathi, and Xinming Ou. A Two-Tier Technique for Supporting Quantifiers in a Lazily Proof-Explicating Theorem Prover. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005*, LNCS 3440, Springer, 2005.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. 39th Annual Design Automation Conference*, 530–535, ACM, New York, 2001.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures *ACM TOPLAS*, 1(2):245–257, 1979.

[NO80] Greg Nelson and Derek C. Oppen. Fast Decision Procedures Based on Congruence Closure. *Journal of the ACM*, 27(2):356–364, 1980.

[NO03] Robert Nieuwenhuis and Albert Oliveras. Congruence closure with integer offsets. In *10^{th} Int. Conf. on Logics for Programming, AI and Reasoning (LPAR), Almaty, Kazakhstan, September 2003*, LNAI 2850, pp. 78–90, Springer, 2003.

[NO03] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In J. Giesl: *16^{th} International Conference on Term Rewriting and Applications, RTA'05*, LNCS 3467, Springer, 2005.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Decision procedures for SAT, SAT Modulo Theories and Beyond. The BarcelogicTools. In G. Sutcliffe, A. Voronkov, eds.: *Logic for Programming, Artificial Intelligence and Reasoning (LPAR), 12^{th} International Conference, LPAR 2005, Montego Bay, Jamaica, December 2005, Proceedings*, LNCS 3835, pp. 23–46, Springer, 2005.

[NOT05] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories. In *Proc. 11^{th} Int. Conf. on Logics for Programming, AI and Reasoning (LPAR), Montevideo, Uruguay, March 2005*, LNAI 3452, pp. 36–50, Springer, 2005.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* LNCS 2283, Springer, 2002.

[OG98] Chris Okasaki and Andy Gill. Fast Mergeable Integer Maps. *The 1998 ACM SIGPLAN Workshop on ML*, Baltimore, Maryland, pages 77–86, September 1998.

[Oka96] Chris Okasaki. *Purely Functional Data Structures. The Formal Semantics of PVS.* Ph.D. thesis. School of Computer Science, Carnegie Mellon University, 1996.

[PJ03] Simon Peyton-Jones, ed. *Haskell 98 Language and Libraries. The Revised Report.* Cambridge University Press, Cambridge, 2003.

[Rob65] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[RS01] Harald Ruess and Natarajan Shankar. Deconstructing Shostak. In *Logic in Computer Science (LICS 2001)*, IEEE Press, 2001.

[Sho67] Joseph R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, Massachusetts, 1967.

[Sim99] Stephen G. Simpson. *Subsystems of Second Order Arithmetic.* Springer, 1999.

[Smu68] Raymond M. Smullyan. *First-Order Logic.* Springer, Berlin, Heidelberg, New York, 1968.

[SORS01] Natarajan Shankar, Sam Owre, John M. Rushby, and David W. J. Stringer-Calvert. *PVS Prover Guide. Version 2.4.* Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001.

[Tar75] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

[TH96] Cesare Tinelli and Mehdi Harandi. A New Correctness Proof of the Nelson–Oppen Combination Procedure. In *Proc. Frontiers of Combining Systems*, ALS 3, 103–120, Kluwer, 1996.

[TS00] Anne S. Troelstra and Helmut Schwichtenberg. *Basic Proof Theory. Second edition.* Cambridge University Press, 2000.

[Vod01a] Paul J. Voda. *Metamathematics of Computer Programming.* Textbook, preliminary draft. Comenius University, Bratislava, May 2001. Available at http://dai.fmph.uniba.sk/~voda/meta.pdf.

[Vod01b] Paul J. Voda. *Introduction of Pairing into PA.* Unpublished manuscript. Bratislava, November 2001. Available at http://ii.fmph.uniba.sk/cl/oldcourses/lpi2/2003-2004/lect/pcantor.pdf

[Wen02] Markus M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents.* PhD. thesis. Institut für Informatik, Technische Universität München, 2002.