MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 28, NO. 4, 2021
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

A Recursive Inclusion Checker for Recursively Defined Subtypes

H. de Nivelle[1]

DOI: 10.18255/1818-1015-2021-4-414-433

[1]School of Engineering and Digital Sciences, Nazarbayev University, 53 Kabanbay Batyr str., Nur-Sultan 010000, Kazakhstan.

MSC2020: 68Q99,03B70
Research article
Full text in English

Received November 15, 2021
After revision December 1, 2021
Accepted December 8, 2021

We present a tableaux procedure that checks logical relations between recursively defined subtypes of recursively defined types and apply this procedure to the problem of resolving ambiguous names in a programming language. This work is part of a project to design a new programming language suitable for efficient implementation of logic. Logical formulas are tree-like structures with many constructors having different arities and argument types. Algorithms that use these structures must perform case analysis on the constructors, and access subtrees whose type and existence depend on the constructor used. In many programming languages, case analysis is handled by matching, but we want to take a different approach, based on recursively defined subtypes. Instead of matching a tree against different constructors, we will classify it by using a set of disjoint subtypes. Subtypes are more general than structural forms based on constructors, we expect that they can be implemented more efficiently, and in addition can be used in static type checking. This makes it possible to use recursively defined subtypes as preconditions or postconditions of functions. We define the types and the subtypes (which we will call adjectives), define their semantics, and give a tableaux-based inclusion checker for adjectives. We show how to use this inclusion checker for resolving ambiguous field references in declarations of adjectives. The same procedure can be used for resolving ambiguous function calls.

Keywords: programming language design; type systems; theorem proving; compiler construction

INFORMATION ABOUT THE AUTHORS

Hans de Nivelle correspondence author	orcid.org/0000-0001-9343-6679. E-mail: hans.denivelle@nu.edu.kz Associate Professor, Dr. Habil.

Funding: Faculty Development Competitive Research Grant Program (FDCRGP) 021220FD1651.

For citation: H. de Nivelle, "A Recursive Inclusion Checker for Recursively Defined Subtypes", *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 414-433, 2021.

Рекурсивная проверка включения для рекурсивно определенных подтипов

Х. де Нивелле[1]

DOI: 10.18255/1818-1015-2021-4-414-433

[1]Школа инженерии и цифровых наук, Назарбаев Университет, ул. Кабанбай Батыра, д. 53, г. Нур-Султан, 010000 Казахстан.

УДК 510.57
Научная статья
Полный текст на английском языке

Получена 15 ноября 2021 г.
После доработки 1 декабря 2021 г.
Принята к публикации 8 декабря 2021 г.

Мы представляем табличную процедуру, которая проверяет логические отношения между рекурсивно определенными подтипами рекурсивно определенных типов, и применяем эту процедуру к проблеме разрешения неоднозначных имен в языке программирования. Эта работа является частью проекта по разработке нового языка программирования, подходящего для эффективной реализации логики. Логические формулы представляют собой древовидные структуры со множеством конструкторов, имеющих различные свойства и типы аргументов. Алгоритмы, использующие эти структуры, должны выполнять анализ вариантов для конструкторов и получать доступ к поддеревьям, тип и существование которых зависят от используемого конструктора. Во многих языках программирования анализ прецедентов обрабатывается путем сопоставления, но мы хотим использовать другой подход, основанный на рекурсивно определенных подтипах. Вместо сопоставления дерева с различными конструкторами мы будем классифицировать его с помощью набора непересекающихся подтипов.
Подтипы являются более общими, чем структурные формы, основанные на конструкторах, мы ожидаем, что они могут быть реализованы более эффективно и, кроме того, могут использоваться при статической проверке типов. Это позволяет использовать рекурсивно определенные подтипы в качестве предварительных условий или постусловий функций. Мы определяем типы и подтипы (которые мы будем называть прилагательными), определяем их семантику и даем проверку включения прилагательных на основе таблиц. Мы показываем, как использовать эту проверку включения для разрешения неоднозначных ссылок на поля в объявлениях прилагательных. Та же процедура может быть использована для разрешения неоднозначных вызовов функций.

Ключевые слова: проектирование языков программирования; системы типов; доказательство теорем; построение компилятора

ИНФОРМАЦИЯ ОБ АВТОРАХ

Ханс де Нивелле автор для корреспонденции	orcid.org/0000-0001-9343-6679. E-mail: hans.denivelle@nu.edu.kz Ассоциированный профессор, доктор физ.-мат. наук.

Финансирование: Программа поддержки исследовательской деятельности профессорско-преподавательского состава 021220FD1651.

Для цитирования: H. de Nivelle, “A Recursive Inclusion Checker for Recursively Defined Subtypes”, *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 414-433, 2021.

1. Introduction

Our goal is to develop a programming language for implementation of logic that is convenient to use on one hand, and efficient on the other hand. Traditionally, functional languages like OCaml ([1]), Haskell ([2]) or Scala ([3]) are considered the most suitable for implementation of logic.

Functional languages have inductive types, and use matching for accessing subtrees. Matching can be viewed as simultaneously inspecting a tree and extracting subtrees into local variables. In our proposed language we want to keep inductive types, but add a mechanism for definition of subtypes, which we will use both as a replacement for matching, and as a basis for static overload resolution. The subtypes make it possible to replace matching by usual field access as used in imperative languages like Java or C^{++}. We believe that field access will be more efficient than matching.

We want our language also to support static overloading of function names. In C, defined functions always must have distinct names. In C^{++} and Java, the same name can be reused for different argument types. The compiler will select the definition that fits best to the arguments. This process is called *overload resolution*. In this paper, we study overload resolution only for field names (which is already present in C). The same technique is applicable to function calls.

We want some imperative features, because efficient run time handling of containers (also called *collections*) is difficult in functional setting. In order to obtain this, data must have efficient low level representation in memory. We allow a restricted form of assignment, only to top level variables and substructures at nesting depth one. In this way, side effects due to sharing can be avoided.

We will start by explaining the subtype system. At run time, it is used for inspecting data (formulas) so that we know which fields exist. The advantage of using subtypes for run time classification is that, once one has the mechanism for defining and verifying them, they are more general than structure matching, and can be used at many other points in the programming language, also at compile time.

As an example of our approach, suppose one has a propositional formula f, and one wants to know if it constructed by **and**. One can either match it into $\mathbf{and}(F_1, F_2)$, or define a subtype 'constructed by **and**', and define two fields f_1, f_2 that can be accessed only on formulas refined by the subtype 'constructed by **and**'. The advantage of subtypes is that they can be easily combined by Boolean operators. For example, it is easy to define the subtype 'constructed by **and** or **or**', or 'atom' and use this as a condition in case analysis.

We will call the subtypes *adjectives*. In addition to adjectives that correspond to simple structural matchings, our adjective system also allows recursively defined subtypes. This makes it possible to define for example negation normal form or conjunctive normal form as a subtype of formulas. Although such recursively defined subtypes can be used for run time classification, they are not intended for this purpose, because evaluating them at run time would be costly. They are intended to be used for overload resolution at compile time. If one restricts oneself to adjectives of constant depth (like the ones above), they can be efficiently checked in constant time at run time.

Adjectives are somewhat similar to liquid types ([4]) or refinement types ([5]) but they have a different aim. Adjectives are not intended for checking operations whose safety depends on arithmetic. Because of this, adjectives do not need SMT solving. We prefer the term *adjective* over *subtype* or *refinement type*, because adjectives can overlap, and are frequently created for one time use. We also want to use adjectives for static type checking and overload resolution. In order to show that this is possible, we define a procedure that resolves ambiguous field references in the adjectives themselves.

We start by defining the type and adjective system in this section. In Section 2 we give the semantics, which can be used for evaluating adjectives ondata, so that they can be used as a replacement for matching. In Section 3, we give a tableaux-based procedure for deciding satisfiability of adjectives. This procedure can be used for checking exhaustiveness of a given case analysis, for checking exclusiveness of cases, and for static type checking. In Section 4 we will apply the tableaux procedure on the problem of resolving ambiguous field references in the adjective declarations themselves. A similar algorithm could be used for resolving ambiguous function calls. We define primitive types:

Definition 1.1. *The primitive types are* bool, char, nat, double, *and* selector. *We assume that all primitive types, with the exception of* selector, *have an order* $<$ *defined on them.*

Type selector is a global enumeration type that consists of named constants. It is intended for defining subtypes (see Definition 1.2 below), or representing logical operators. Actually, these uses are the same because the logical operator of a formula defines a subtype at the same time. Enumeration types in Rust ([6]) play a similar role. We write elements of selector as identifiers preceded by a question mark, e.g. `?and,?or,?implies,?equiv`, or `?zero,?succ`. We give examples of their use below, after the definition of the type system.

Next we define non-primitive types, and after that follow adjectives. Strictly seen, the definition of adjectives should come first, because types contain adjectives while adjectives do not contain types. We find, however, that this order would be unintuitive, because it would be hard to understand the usefulness of adjectives, without having seen the definition of types. Because of this, we start by introducing types:

Definition 1.2. Simple types *are recursively defined as follows:*

- *A primitive type is a simple type.*
- *An identifier is a simple type (assuming that it has been defined as type, see Definition 1.4 below).*
- *If T is a simple type, A is an adjective, then $T \circ A$ is also a simple type.*

We recursively define compound types*:*

1. *If $v_1, \ldots, v_n$ are identifiers, $V_1, \ldots, V_n$ are simple types with $n \geq 0$, then $(\ v_1\colon V_1, \ldots, v_n\colon V_n\ )^*$ is a compound type.*
2. *If C is a compound type, v is an identifier and V is a simple type, then $v\colon V,\ C$ is a compound type as well.*
3. *If $C_1, \ldots, C_m$ with $m \geq 1$ are compound types, $A_1, \ldots, A_m$ are adjectives, s is an identifier, then $s?(A_1 \Rightarrow C_1, \ldots, A_m \Rightarrow C_m)$ is a compound type as well.*

In case 1, the fields $v_1, \ldots, v_n$ are called repeated *fields. All other fields defined in cases 2 and 3 are called* scalar fields. *In case 3, idenifier s (which must refer to an earlier defined field) is called the* pivot *field. Case 3 defines compound types where the existence of later fields depends on the adjective that applies to s.*

We distinguish between simple and compound types, because we want every compound type to have a name, which solves some technical problems. It has no consequences for expressiveness, because one can always define a compound type, and use the name as simple type. This can be automatically done by the compiler, if needed. The meaning of $T \circ A$ is: Type T refined by adjective A.

Compound types have a tree structure, where every path is a possible realization of the type. The tree branches whenever case 3 is used. In that case, identifier s should refer to a field that occurs before it on the same path. We cannot impose this in Definition 1.2, because trees are constructed bottom up, and the field s must be defined above it in an application of case 2. We give a simple example of a compound type:

Example 1.3.

$$T = \quad s\colon \text{selector}\circ(?\text{simp}|?\text{vect}),\ s?(\ \ ?\text{simp} \Rightarrow a\colon A,\ b\colon B,\ ()^*, \qquad ?\text{vect} \Rightarrow c\colon C,\ (d\colon D,\ e\colon E\,)^* \quad)$$

Example 1.3 is purely technical. Type T is not intended to be meaningful. Also note that Definition 1.2 is awkward to use when defining types in practice because it is intended as technical representation. It is not intended as the syntax with which the programmer will define types. The first field s is a selector, which must be either ?simp or ?vect. In case it is ?simp, there are two more scalar fields a of type A, and b of type B. In case s equals ?vect, there is one more scalar field c of type C, and an unbounded number of repeated fields

d of type D and e of type E. They must be accessed with array notation $d[i]$ or $e[i]$. They are implemented as vectors in C^{++}, which means that the repeated part can dynamically grow and shrink, and an object of type T will be reallocated when it runs out of capacity. Because repeated fields are always last, offsets of scalar fields do not depend on the number of repeated fields present.
For the moment, we assume that different fields of different types have distinct names. This is an unrealistic restriction for a real programming language, because field names can be reused in different types. In Section 4 we will allow reuse of field names between different options of a type, and between different types. We will give a procedure that is able to resolve such ambiguous field references into unambiguous references.

Type definitions need to be stored in a mapping. It is convenient to use two distinct mappings, one for the simple types, and one for the compound types:

Definition 1.4. *We define two mappings Σ_S and Σ_C. The first mapping maps identifiers to simple types, and the second mapping maps identifiers to compound types. If Σ_S contains a value for v, we write $\Sigma_S(v)$ for the value. Similarly, if Σ_C contains a value for v, we write $\Sigma_C(v)$ for the value. We assume that Σ_S and Σ_C do not contain a value for the same variable v. We also assume that Σ_S is cycle free.*

An example of Σ_S not being cycle-free would be when $\Sigma_S(v_1) = v_2 \circ A_2$, and $\Sigma_S(v_2) = v_1 \circ A_1$. Cycles inside Σ_C are unproblematic, see for example the definition of prop below in Example 1.10.

This completes the definition of the types, next we define adjectives:

Definition 1.5. *We recursively define adjectives, starting with primitive adjective constructors:*

- *If c is a constant of one of the primitive types* bool, char, nat, double, *then c and $c^{\geq}$ are adjectives.*
- *If c is a constant of type* selector, *then c is an adjective.*
- **empty** *is an adjective.*
- *If v is an identifier, then v is an adjective.*

Next we define recursive adjective constructors:

- *If f is an identifier, and A is an adjective, then $f(A)$ is an adjective.*
- *If A is an adjective, then* **first**(A) *and* **rest**(A) *are adjectives.*
- *If A is an adjective, then $\forall A$ and $\exists A$ are adjectives.*

Adjectives of any type can be combined by propositional operators:

- *If $A_1, \dots, A_n$ are adjectives, then $A_1 \vee \cdots \vee A_n$, and $A_1 \wedge \cdots \wedge A_n$ are also adjectives.*
- *If A is an adjective, $\neg A$ is also an adjective.*

The intuitive meaning of the adjective c is : equal to c. The intuitive meaning of $c^{\geq}$ is : greater or equal to c. The intuitive meaning of $f(A)$ is: whose f-field satisfies A. If f is a repeated field, then A must be equal to **empty** or have one of the forms $\forall A$, $\exists A$, **first**(A), or **rest**(A). The meaning of **empty** is that no repeated fields are present. The meaning of $f(\forall A)$ is that all repetitions of f must satisfy A. Similarly $f(\exists A)$ means that at least one repetition of f must satisfy A. The meaning of **first**(A) is that $f[1]$ must satisfy A. The meaning of **rest**(A) is that all repetitions of f, except for possibly the first, must satisfy A.

Definition 1.6. *We assume a mapping Σ_A that maps identifiers to pairs of form (T, A), where T is a simple type, and A is an adjective. If Σ_A contains a value for v, we write $\Sigma_A(v)$ for the value.*

The intuitive meaning of $\Sigma_A(v) = (T, A)$ is: Identifier v is defined on type T as adjective A.

Definition 1.7. *We write $A[v]$ for an adjective that contains v somewhere not inside a subformula of form $f(\cdot)$. We define a* cycle *as a sequence of identifiers, s.t.*

$$\Sigma_A(v_1) = (\,T_2, A_2[v_2]\,), \dots, \Sigma_A(v_{n-1}) = (\,T_n, A_n[v_n]\,),\ \Sigma_A(v_n) = (\,T_1, A_1[v_1]\,).$$

Cycles are logically problematic when they involve negation. Because the intended meaning of an adjective is an inductively defined predicate, an adjective defined through a cycle involving negation, would be ill-defined. This is similar to the situation in logic programming ([7]). Although it would be possible to adapt Definitions 2.2 and 3.8 to monotonic cycles, we are not aware of a meaningful use of it. Therefore, we think it is better to forbid cycles altogether. In the rest of this paper, we will assume that Σ_A is cycle free. We give some examples of type and adjective definitions:

Example 1.8. *The type of complex numbers can be defined as follows:*

$$\text{complex} := \text{re}\colon \text{double},\ \text{im}\colon \text{double},\ (\,)^*.$$

Complex numbers have two scalar fields re *and* im, *and no repeated fields.*

Example 1.9. *Natural numbers can be defined as follows:*

$$\text{nat} := \text{sel}\colon \text{selector}, \text{sel?}\begin{pmatrix} \text{?zero} & \Rightarrow & (\,)^* \\ \text{?succ} & \Rightarrow & \text{pred}\colon \text{nat},\ (\,)^* \end{pmatrix}$$

Adjectives odd and even can be defined on nat *in mutual recursion:*

$$\begin{array}{lll} \text{even} & := & (\,\text{nat},\ \text{sel(?zero)} \vee (\,\text{sel(?succ)} \wedge \text{pred(odd)}\,)\,) \\ \text{odd} & := & (\,\text{nat},\ \text{sel(?succ)} \wedge \text{pred(even)}\,) \end{array}$$

Example 1.10. *We define propositional logic:*

$$\text{prop} := \text{op}\colon \text{selector},\ \text{op?}\begin{pmatrix} \text{?var} & \Rightarrow & (\,c\colon \text{char}\,)^* \\ \text{?not} & \Rightarrow & \text{sub}\colon \text{prop},\ (\,)^* \\ \text{?implies}\vee\text{?equiv} & \Rightarrow & \text{sub}_1\colon \text{prop},\ \text{sub}_2\colon \text{prop},\ (\,)^* \\ \text{?and}\vee\text{?or} & \Rightarrow & (\,\text{sub}_\text{n}\colon \text{prop}\,)^* \end{pmatrix}$$

There is no need to define $\top$ or $\bot$ separately, because $\top = \bigwedge \emptyset$, and $\bot = \bigvee \emptyset$. A variable consists of ?var combined with a finite number of characters. A negation consists of ?not combined with a single subformula called sub. A conjunction or disjunction consists of ?and or ?or, followed by an arbitrary number of subformulas called sub_n. We define a few adjectives on prop:

Example 1.11. *'Constructed by* ?and*' can be expressed as* op(?and). *'Constructed by* ?and *or* ?or*' can be expressed as* op(?and∨?or). *The property of being an atom can be expressed by*

$$\text{atom} := (\,\text{prop}, \text{op(var)}\,).$$

The property of being a literal can be expressed by

$$\text{literal} := (\,\text{prop}, \text{atom} \vee (\,\text{op(?not)} \wedge \text{sub(?atom)}\,)\ \,).$$

Negation normal form (NNF) can be expressed by:

$$\text{nnf} := (\,\text{prop},\ \bigvee \left\{ \begin{array}{l} \text{literal} \\ \text{op(?and}\vee\text{?or)} \wedge \text{sub}_\text{n}(\,\forall \text{nnf}\,) \end{array} \right. \ ).$$

Conjunctive normal form (CNF) can be defined as follows:

$$\text{cnf} := (\,\text{prop}, \text{op(?or)} \wedge \text{sub}_\text{n}(\forall(\text{op(?and)} \wedge \text{sub}_\text{n}(\,\forall \text{literal}\,)))\,).$$

Case 3 in Definition 1.2 introduces fields whose existence depends on the adjectives fulfilled by the pivot s. For type prop, the sub field only exists when the op field equals ?not, which can be expressed by the adjective op(not). Similarly, field sub_1 exists only when op(?implies$\vee$?equiv). The following definition specifies how to obtain preconditions from the definition of the compound type.

Definition 1.12. *Let f be a exact identifier that is declared as field in a compound type C. We recursively define the* precondition *of f, written as* $\text{PREC}(C, f)$, *as follows:*

- *if C has form* $(\ v_1 : V_1, \dots, v_n : V_n\ )^*$, *then* $\text{PREC}(f, C) = \top$.
- *If C has form $v\colon V,\ C'$, and $f = v$, then* $\text{PREC}(f, C) = \top$. *If f is declared in C', then* $\text{PREC}(f, C) = \text{PREC}(f, C')$.
- *If C has form $s?(A_1 \Rightarrow C_1, \dots, A_m \Rightarrow C_m)$, and $f = s$, then* $\text{PREC}(f, C) = A_1 \vee \dots \vee A_m$. *Otherwise, f must be declared in exactly one C_j. We define* $\text{PREC}(f, C) = s(A_j) \wedge \text{PREC}(f, C_j)$.

Since f can occur in Σ_C only once, it is possible to write $\text{PREC}(f)$ *instead of* $\text{PREC}(f, C)$, *where C is the compound type that contains f.*

2. Semantics of Types and Adjectives

We will describe the semantics of adjectives and types using a simplified high level representation of data. We assume that data are represented by trees whose subtrees are labeled with field names. This representation is convenient for defining the semantics without having to consider low level details like memory lay out. In the implementation we use a low level representation, where scalar fields have a fixed offset, and repeated fields have an offset that can be computed by multiplying the index with the size of the repeated part, and adding a base offset.

Definition 2.1. *We define the set of data trees D, and the set $\overline{D}$ of finite sequences of data trees, in simultaneous recursion:*

- *If d is an element of one of the primitive types T, defined in Definition 1.1, then $d \in D$.*
- *If $d_1, \dots, d_n$ $(n \geq 0)$ is a finite sequence of data trees, then $(d_1, \dots, d_n) \in \overline{D}$. We will write $\|\overline{D}\|$ for the length n of $\overline{D}$.*
- *If $f_1, \dots, f_n$ are pairwise distinct identifiers, and each $d_i \in D \cup \overline{D}$, then $\{(f_1, d_1), \dots, (f_n, d_n)\} \in D$.*

Data trees of the third type can be viewed as partial functions mapping each f_i to d_i, such that d_i is either a data tree or a sequence of data trees. When d is of the third type, we write $d.f$ for the value attached to f when it exists. If $d.f \in \overline{D}$, we write $d.f[i]$ for its i-th element, assuming that $i \leq \|d.f\|$.

We will define when a data tree has a given type, and when a data tree (or sequence of data trees) satisfies a given adjective. Types may contain adjectives, but adjectives do not contain types. Therefore, we start by defining when a data tree (or sequence of data trees) makes an adjective true:

Definition 2.2. *We define in simultaneous recursion when a data tree makes an adjective true, and when a sequence of data trees makes an adjective true. We use the notation $d \models A$ for both cases.*
We first consider the cases where $d \in D$:

- *If c is a constant, then $d \models c$ iff $d = c$, and $d \models c^{\geq}$ if $d \geq c$.*
- *If f is an identifier, s.t. $d.f$ exists, then*
  - *if $d.f$ is a single data tree, then $d \models f(A)$ iff $d.f \models A$.*
  - *if $d.f$ is a sequence of data trees, $d \models f(A)$ iff $d.f \models A$.*

  The two cases appear to be the same, but $d.f$ have different types, hence we prefer to list them separately.
- *If v is an identifier, s.t. $\Sigma_A(v)$ is defined, and $d \in D$, then $d \models v$ iff $d \models \Sigma_A(v)$.*

Next we list the cases where $d \in \overline{D}$:

- $(d_1, \dots, d_n) \models$ **empty** *iff* $n = 0$.
- $(d_1, \dots, d_n) \models \forall A$ *iff for every d_i we have $d_i \models A$.*

- $(d_1, \dots, d_n) \models \exists A$ *iff there exists a* d_i *such that* $d_i \models A$.
- $(d_1, \dots, d_n) \models \mathbf{first}(A)$ *iff* $n \geq 1$ *and* $d_1 \models A$.
- $(d_1, \dots, d_n) \models \mathbf{rest}(A)$ *iff* $n \geq 1$ *and* $(d_2, \dots, d_n) \models A$.

The definitions for the propositional connectives are standard, both for $d \in D$ *and* $d \in \overline{D}$:

- $d \models A_1 \vee \cdots \vee A_n$ *if there is an* i $(1 \leq i \leq n)$, *s.t.* $d \models A_i$.
- $d \models A_1 \wedge \cdots \wedge A_n$ *for all* i $(1 \leq i \leq n)$, *one has* $d \models A_i$.
- $d \models \neg A$ *iff not* $d \models A$.

We define when a tree has a certain type:

Definition 2.3. *We recursively define when a data tree* d *has type* T. *We use notation* $d\colon T$. *We first list the cases where* T *is simple:*

- *For a primitive type* T, *we define* $d\colon T$ *as* $d \in T$.
- *If* v *is an identifier that is defined in* Σ_S, *then* $d\colon v$ *iff* $d\colon \Sigma_S(v)$.
- *If* v *is an identifier that is defined in* Σ_C, *then* $d\colon v$ *iff* $d\colon \Sigma_C(v)$.
- $d\colon (T \circ A)$ *iff* $d\colon T$ *and* $d \models A$.

Next we list the cases where T *is compound:*

- $d\colon (\ v_1\colon V_1, \dots, v_n\colon V_n\ )^*$ *iff either*
  - $n = 0$, *or*
  - $n > 0$ *and* $d.v_1, \dots, d.v_n$ *are all defined, are all in* $\overline{D}$, *have the same length* $L = \|d.v_1\| = \cdots = \|d.v_n\|$, *and for every* i $(1 \leq i \leq n)$ *and* j $(1 \leq j \leq L)$, *we have* $d.v_i[j]\colon V_i$.
- $d\colon (v\colon V,\ C)$ *iff* $d.v$ *is defined,* $d.v\colon V$, *and* $d\colon C$.
- $d\colon s?(A_1 \Longrightarrow C_1, \dots, A_m \Longrightarrow C_m)$ *iff* $d.s$ *is defined, in* D, *and there is exactly one* $1 \leq i \leq m$, *s.t.* $d.s \models A_i$ *and* $d\colon C_i$.

Example 2.4. *Following up on Example 1.8, we can see that* $\{\ (\mathrm{re}, 1.0), (\mathrm{im}, 2.0)\ \}$ *has type* complex.

Example 2.5. *Using the declaration of* prop *in Example 1.10, the propositional formula* p *will be represented by a data tree* d_p *with fields*

$$\begin{array}{lcl} d_p.\mathrm{op} & = & ?\mathrm{atom} \\ d_p.\mathrm{c} & = & ('p') \end{array}$$

The propositional formula q *will be represented by* d_q *with fields*

$$\begin{array}{lcl} d_q.\mathrm{op} & = & ?\mathrm{atom} \\ d_q.\mathrm{c} & = & ('q') \end{array}$$

Similarly, the propositional formula r *will be represented by* d_r *with*

$$\begin{array}{lcl} d_r.\mathrm{op} & = & ?\mathrm{atom} \\ d_r.\mathrm{c} & = & ('r') \end{array}$$

The formula $q \vee r$ *will be represented by* d *with*

$$\begin{array}{lcl} d.\mathrm{op} & = & ?\mathrm{or} \\ d.\mathrm{sub} & = & (d_q, d_r) \end{array}$$

Finally, the formula $p \longrightarrow (q \vee r)$ *will be represented by* d' *with*

$$\begin{array}{lcl} d.\mathrm{op} & = & ?\mathrm{implies} \\ d.\mathrm{sub}_1 & = & d_p \\ d.\mathrm{sub}_2 & = & d' \end{array}$$

3. A Tableaux Calculus

In this section we define a calculus for checking satisfiability of adjectives. This is sufficient to answer all logical questions that are needed for the implementation of our programming language. In order to show that adjectives $A_1, \ldots, A_n$ cover all possible cases in a switch, it is sufficient to show that $A \wedge \neg A_1 \wedge \cdots \wedge \neg A_n$ is unsatisfiable, where A is the adjective part of the type of the switch expression. In order to show that a switch has no overlapping cases, it is sufficient to show that $A \wedge A_i \wedge A_j$ is unsatisfiable for all distinct i and j. In order to show that a function f_1 defined on adjective A_1 is a better fit than function f_2 defined on adjective A_2, it is sufficient to show that $A_1 \wedge \neg A_2$ is unsatisfiable, while $A_2 \wedge \neg A_1$ is satisfiable.

In order to use our calculus, types need to be decomposed into their adjective part, and the part that specifies their implementation. The calculus needs the adjective part of a type, but does not use the implementation. We will define two functions that decompose a simple type into their adjective and implementation components.

Definition 3.1. *An* implementation type *is either a primitive type, or an identifier defined in* Σ_C*, i.e. the name of a compound type. For a simple type* T*, we define* $\mathrm{ADJ}(T)$ *and* $\mathrm{IMPL}(T)$ *as follows:*

- *If* T *is primitive then* $\mathrm{ADJ}(T) = \top$*, and* $\mathrm{IMPL}(T) = t$*.*
- *If* v *is an identifier defined in* Σ_C*, then* $\mathrm{ADJ}(v) = \top$*, and* $\mathrm{IMPL}(v) = v$*.*
- $\mathrm{ADJ}(T \circ A) = \mathrm{ADJ}(T) \wedge A$*, and* $\mathrm{IMPL}(T \circ A) = \mathrm{IMPL}(T)$*.*

For example, if $T = \mathrm{prop} \circ \mathrm{nnf} \circ \mathrm{cnf}$, then $\mathrm{ADJ}(T) = \mathrm{nnf} \wedge \mathrm{cnf}$, and $\mathrm{IMPL}(T) = \mathrm{prop}$.

Theorem 3.2. *If* T *is a simple type, and* d *is a data tree, then* $d\colon T$ *iff* $d\colon \mathrm{IMPL}(T)$ *and* $d \models \mathrm{ADJ}(T)$*.*

The theorem can be easily verified by applying the rules of Definition 2.2.

At this point, we can define a simple interface of our tableaux procedure. Its input is just a single type T. The procedure establishes that there exists no data term d with type T. The applications that we mentioned in the introduction, can be obtained as follows:

- Establish that there exists no data term d of type T that satisfies adjective A : Call the procedure with $T \circ A$.
- Establish that every data term d of type T, that satisfies A, must also satisfy B : Call the procedure with $T \circ A \circ \neg B$.

The procedure works on the following normal form, which is similar to negation normal form:

Definition 3.3. *We recursively define when an adjective is in* path normal form *(PNF):*

- *If* c *is a constant of a primitive type, then* c*,* $\neg c$*,* $c^{\geq}$ *and* $\neg c^{\geq}$ *are in PNF. In the last two cases,* c *cannot have type* selector.
- **empty** *and* $\neg$**empty** *are in PNF.*
- *If* v *is an identifier, then* v *and* $\neg v$ *are in PNF.*
- *If* A *is in PNF, then* $f(A)$*,* $\mathbf{first}(A)$*,* $\mathbf{rest}(A)$*,* $\forall A$*, and* $\exists A$ *are in PNF.*
- *If* $A_1, \ldots, A_n$ $(n \geq 0)$ *are in PNF, then* $A_1 \vee \cdots \vee A_n$ *and* $A_1 \wedge \cdots \wedge A_n$ *are in PNF.*

Definition 3.4. *An adjective* A *can be brought into PNF by calling* $\mathrm{PNF}(A, \mathbf{pos})$*, where* $\mathrm{PNF}(A, p)$ *is defined by cases as follows:*
If A *has form* c*,* $c^{\geq}$*,* **empty***, or* v*, then*

$$\mathrm{PNF}(A, \mathbf{pos}) = A$$
$$\mathrm{PNF}(A, \mathbf{neg}) = \neg A$$

For the remaining cases:

$$\begin{aligned}
&\mathrm{PNF}(A_1 \vee \cdots \vee A_n, \mathbf{pos}) = \mathrm{PNF}(A_1, \mathbf{pos}) \vee \cdots \vee \mathrm{PNF}(A_n, \mathbf{pos})\\
&\mathrm{PNF}(A_1 \vee \cdots \vee A_n, \mathbf{neg}) = \mathrm{PNF}(A_1, \mathbf{neg}) \wedge \cdots \wedge \mathrm{PNF}(A_n, \mathbf{neg})\\
&\mathrm{PNF}(A_1 \wedge \cdots \wedge A_n, \mathbf{pos}) = \mathrm{PNF}(A_1, \mathbf{pos}) \wedge \cdots \wedge \mathrm{PNF}(A_n, \mathbf{pos})\\
&\mathrm{PNF}(A_1 \wedge \cdots \wedge A_n, \mathbf{neg}) = \mathrm{PNF}(A_1, \mathbf{neg}) \vee \cdots \vee \mathrm{PNF}(A_n, \mathbf{neg})\\
&\mathrm{PNF}(f(A), p) = f(\,\mathrm{PNF}(A, p)\,) \textit{ for } p \in \{\mathbf{pos}, \mathbf{neg}\}\\
&\mathrm{PNF}(\mathbf{first}(A), p) = \mathbf{first}(\,\mathrm{PNF}(A, p)\,) \textit{ for } p \in \{\mathbf{pos}, \mathbf{neg}\}\\
&\mathrm{PNF}(\mathbf{rest}(A), p) = \mathbf{rest}(\,\mathrm{PNF}(A, p)\,) \textit{ for } p \in \{\mathbf{pos}, \mathbf{neg}\}\\
&\mathrm{PNF}(\exists A, \mathbf{pos}) = \exists\,\mathrm{PNF}(A, \mathbf{pos})\\
&\mathrm{PNF}(\exists A, \mathbf{neg}) = \forall\,\mathrm{PNF}(A, \mathbf{neg})\\
&\mathrm{PNF}(\forall A, \mathbf{pos}) = \forall\,\mathrm{PNF}(A, \mathbf{pos})\\
&\mathrm{PNF}(\forall A, \mathbf{neg}) = \exists\,\mathrm{PNF}(A, \mathbf{neg})\\
&\mathrm{PNF}(\neg A, \mathbf{pos}) = \mathrm{PNF}(A, \mathbf{neg})\\
&\mathrm{PNF}(\neg A, \mathbf{neg}) = \mathrm{PNF}(A, \mathbf{pos})
\end{aligned}$$

The following theorem can be straightforwardly proven, using Definition 2.2:

Theorem 3.5. *Let A be an adjective. For every data tree d, we have $d \models A$ iff $d \models \mathrm{PNF}(A, \mathbf{pos})$. Similarly, $d \models \neg A$ iff $d \models \mathrm{PNF}(A, \mathbf{neg})$.*

The path normal form of $\neg(A \vee f(\mathbf{rest}(\forall\neg B)))$ equals $\neg A \wedge f(\mathbf{rest}(\exists B))$.

Definition 3.6. *We define* a path *as a finite sequence $(f_1, \ldots, f_n)$ with $(n \geq 0)$, where each f_i is either an identifier (representing a field name), or an element of $\{\exists, \forall, \mathbf{first}, \mathbf{rest}\}$. We write ϵ for the empty path, and use notation $\pi.f$ for extending path π with f. If path π' can be obtained from path π by zero or more extensions, we call π a prefix of π'. We call π a* strict prefix *of π' if at least one extension was used.*

We assume that there exists a total order $<$ on paths with the property that if π_1 is a strict prefix of π_2, then $\pi_1 < \pi_2$. Such an order can be easily obtained by fixing a total order on all possible f_i, and using the alphabetic, lexicographic extension.

Definition 3.7. *An* adjective stack *S is a finite sequence of triples (π_i, A_i, λ_i), where each π_i is a path, each A_i is an adjective, and each $\lambda_i \in \mathcal{N}$. We write $S[\pi]$ for the set $\{\, A \mid \exists\lambda$ s.t. (π, A, λ) occurs in $S \,\}$ and $S_\lambda[\pi]$ for the set $\{\, A \mid (\pi, A, \lambda)$ occurs in $S \,\}$.*

The attribute λ_i stores the level of A_i on path π_i. More precisely, an adjective that was obtained from a formula on a strict prefix of π_i will have level 0. An adjective that was obtained from an adjective with level λ on the same path, receives level $\lambda + 1$.

We define the tableaux procedure. In order to obtain termination, the procedure uses a blocking rule. Whenever a new path π is entered, it checks that there is no strict prefix π' of π containing a set of adjectives that are included in the formulas on the current path. For example, if path (f) contains A, B, and path (f, g) contains formulas A, B, C, we can close the branch. This rule is correct, because for every data tree d, the subtree $d.f.g$ is a subtree of $d.f$. If it is possible that $d.f.g \models A \wedge B \wedge C$, then one can replace $d.f$ by $d.f.g$ and obtain a smaller data tree, for which $d.f \models A \wedge B$.

The tableaux procedure always tries to extend in deterministic fashion first. If that does not result in a conflict, it selects a path π and tries all possible non-deterministic choices on this path. It keeps on trying until it either has explored all choices, or obtained a consistent stack S.

Definition 3.8. *We define a procedure that tries to establish that no data term d can have type T. The procedure starts by creating a stack with one element:*

$$S = (\,(\epsilon, \mathrm{ADJ}(T), 0)\,).$$

After that, it calls $b =$ deterministic(0). *If* $b = \bot$, *it returns* $\bot$. *Otherwise, it returns* **nondeterministic**(ϵ).
We list the subprocedures, starting with the procedure **deterministic**(d). *All subprocedures have access to the stack* S. *We write* $\|S\|$ *for the size of the stack, and write the i-th element in the form* $S_i = (\pi_i, A_i, \lambda_i)$.
Procedure **deterministic**(d) *must be called with a natural number* $1 \leq d \leq \|S\|$. *It returns* $\bot$ *or* $\top$, *with* $\bot$ *indicating contradiction. The implementation is as follows:*

1. *Set* $c = d$.
2. *If* $c \leq \|S\|$, *then*
   - *If* A *has form* v *with* v *an identifier that has no definition in* Σ_A, *then return* $\bot$.
   - *If* A *has form* $\neg c^{\geq}$ *with* c *the minimal element in its type, then return* $\bot$.
   - *For all* $i < c$ *with* $\pi_i = \pi_c$, *check whether* A_i *and* A_c *are in conflict, using the rules listed below. If they are in conflict, return* $\bot$. *Otherwise try the next* π_i. *The rules are:*
     - *A complementary pair* $A, \neg A$ *is in conflict.*
     - *A pair* c_1, c_2 *of any primitive type with* $c_1 \neq c_2$ *is in conflict.*
     - *A pair of form* $\neg c_1^{\geq}$, $c_2^{\geq}$ *with* $c_1 \leq c_2$ *is in conflict.*
     - *A pair of form* $c_1^{\geq}$, c_2 *with* $c_1 > c_2$, *is in conflict.*
     - *A pair of form* $\neg c_1^{\geq}, c_2$ *with* $c_1 \leq c_2$ *is in conflict.*
   - *Assign* $c = c + 1$ *and go back to step 1.*
3. *If* $d \leq \|S\|$, *check if one of the deterministic extension rules in the table below is applicable on* A_d. *If yes, then for every deterministic consequence* A', *push* $(\pi_d, A', \lambda_d + 1)$ *to* S.

v	$\Rightarrow$	PNF$(A,$ **pos**$)$ *if* Σ_A *contains a definition of form* $v := (T, A)$
$\neg v$	$\Rightarrow$	PNF$(A,$ **neg**$)$ *if* Σ_A *contains a definition of form* $v := (T, A)$
$\forall A$	$\Rightarrow$	**empty** $\vee$ (**first**$(A) \wedge$ **rest**$(\forall A)$)
$\exists A$	$\Rightarrow$	$\neg$**empty** $\wedge$ (**first**$(A) \vee$ **rest**$(\exists A)$)
$A_1 \wedge \cdots \wedge A_n$	$\Rightarrow$	A_i *for each* $1 \leq i \leq n$

4. *If* A_d *has form* $f(A')$, *then push* $(\pi_d.f, A', 0)$ *and* $(\pi_d.f, \mathrm{ADJ}(T), 0)$ *to* S, *where* ADJ(T) *is the adjective component of the type* T *of* f.
5. *Set* $d = d + 1$. *Goto step 1.*

Procedure **nondeterministic**(π) *must be called with a path* π *that occurs in* S. *It returns* $\bot$ *or* $\top$, *with* $\bot$ *indicating contradiction. The implementation is as follows:*

1. *For every strict prefix* π' *of* π, *do:*
   (a) *if* $S_0[\pi'] \subseteq S[\pi]$, *then return* $\bot$. *(This is the blocking rule, mentioned above.)*
2. *Call* **nondeterministic**$(\pi, 1)$.

Procedure **nondeterministic**(π, n) *must be called with a path* π *that occurs in* S, *and with* $1 \leq n \leq \|S\|$. *It returns* $\bot$ *or* $\top$, *with* $\bot$ *indicating contradiction. The implementation is as follows:*

1. *Find the smallest* $n' \geq n$, *such that* $A_{n'}$ *has form* $A_1 \vee \cdots \vee A_m$ *with* $m \geq 2$.
2. *If no* n' *was found in the previous step, then find the next path* $\pi' > \pi$ *occurring in* S, *using the alphabetic lexicographic order* $>$. *Call* $b =$ **nondeterministic**(π') *and return* b. *If no path could be found, return* $\bot$.
3. *Set* $n = n'$. *We know that* S_n *has form* $(\pi,\ A_1 \vee \cdots \vee A_m, \lambda)$, *with* $m \geq 2$. *For i from* 1 *to* n, *do the following:*
   - *Set* $s = \|S\|$. *Push* $(\pi, A_i, \lambda + 1)$ *on the stack.*
   - *Call* $b =$ **deterministic**(s). *If* $b \neq \bot$, *call* $b =$ **nondeterministic**$(\pi, n + 1)$. *If* $b = \top$ *return* $\top$.
   - *Restore* S *to length* s.
4. *Set* $n = n + 1$. *Restart at step 1.*

Procedure **nondeterministic**(π) implements a loop checker which guarantees termination. Its correctness is based on the fact that, if a satisfying data tree can be found, it can be pruned into a data tree in which the calculus does not repeat initial states. We state the following without proof:

Theorem 3.9. *The tableaux calculus of Definition 3.8 terminates.*

In order to establish correctness of Theorem 3.9, it is sufficient establish that every branch is finite. During search, the procedure will only introduce subadjectives of the initial type, combined with subadjectives of identifiers defined in Σ_A. Therefore, blocking must eventually happen.

Theorem 3.10. *The procedure of Definition 3.8 is complete.*

Доказательство. The proof will be a bit informal. We have to prove that if no data tree d with $d\colon T$ exists, then the procedure of Definition 3.8 will reject T. We will prove the converse: If T is not rejected, then there exists a data tree d, s.t. $d\colon T$. Suppose that T is not rejected. This implies that the procedure terminates with an open branch. Let $S = (\pi_1, A_1, \lambda_1), \ldots, (\pi_n, A_n, \lambda_n)$ be the state of the stack with which the tableaux procedure terminated.
For a path occurring in S, let

$$S(\pi) = \{A \mid S \text{ contains a tuple } (\pi_i, A_i, \lambda_i), \text{ s.t. } \pi_i = \pi \text{ and } A_i \text{ has form } c,\ c^{\geq} \text{ or } \neg c^{\geq}\}.$$

For an arbitrary constant c, define

$$\begin{array}{lcl} I(c) & = & \{c\} \\ I(c^{\geq}) & = & \{c' \mid c' \geq c\}, \\ I(\neg c^{\geq}) & = & \{c' \mid c' < c\}. \end{array}$$

It is easily checked that for every π occurring in S, we have $\bigcap\{I(A) \mid A \in S(\pi)\} \neq \emptyset$, because otherwise the branch would be closed. Hence, we can select a constant c_π from each such set. Let A_S be the adjective

$$A_S = \bigwedge\{\pi(c_\pi) \mid \pi \text{ occurs in } S\}.$$

The adjective A_S states that for a given data tree d, selecting field sequence π will result in constant c_π. It is easy to construct a data tree d_S, s.t. $d_S \models A_S$ by starting with the required constants, and combining them as required by A_S. We show by backward induction (that is from n towards 1), that

$$d_S \models \pi_i(A_i) \wedge \cdots \wedge \pi_n(A_n).$$

Assume that we already have established that $d_S \models \pi_{i+1}(A_{i+1}) \wedge \cdots \wedge \pi_n(A_n)$. We proceed by case analysis on the form of A_i. We need to consider only the forms of A_i that do not close the branch.

- if A_i is an identifier v, then we know that v has a definition (T, A') in Σ_A, since otherwise the branch would have been closed. Hence we know that PNF(A', **pos**) occurs among $A_{i+1}, \ldots, A_n$. As a consequence, $d_S \models$ PNF(A', **pos**). By Theorem 3.5, we know that $d_S \models A'$.
- If A_i is a negated identifier $\neg v$, then if v has no definition in Σ_A, we have $d_S \not\models \pi_i(v)$, so that $d_S \models \pi_i(\neg v)$. If v has a definition (T, A') in Σ_A, then PNF(A', **neg**) occurs among $A_{i+1}, \ldots, A_n$. By induction, $d_S \models$ PNF(A', **neg**). By Theorem 3.5, we have $d_S \models \neg A'$.
- The other cases can be obtained by inspecting the cases in Definition 2.2.

□

Theorem 3.11. *The procedure of Definition 3.8 is sound.*

Доказательство. In order to prove soundness, one must prove that if the procedure rejects a type T, then there is no data tree d with $d\colon T$. We will prove the converse: If there is a data tree d with $d\colon T$, then T will not be rejected by the procedure.

This would be trivial, if the blocking rule would not exist. It is easy to show that if $d\colon T$, there exists a stack $S = (\pi_1, A_1, \lambda_1), \ldots, (\pi_n, A_n, \lambda_n)$, with $(\pi_1, A_1, \lambda_1) = (\epsilon, \text{ADJ}(T), 0)$, that represents an open branch of the tableaux procedure when the blocking rule is not used.

Now assume that S will be closed when the blocking rule can be used. We will show that there exists a shorter stack S' which also starts with $S'_1 = (\epsilon, \mathrm{ADJ}(T), 0)$, and which also represents an open branch of the tableaux procedure when the blocking rule cannot be used. Repeating this process will result in a stack that does not contain any more applications of the blocking rule.

Let S be a stack with $S_1 = (\epsilon, \mathrm{ADJ}(T), 0)$, taken from an open branch of the tableaux procedure, when blocking is not used. Suppose that somewhere in S, the blocking rule would be applicable. This means that there exist π and π', s.t. π' is a strict prefix of π, and $S_0[\pi'] \subseteq S[\pi]$. We prune S as follows:

- Remove every (π_i, A_i, λ_i), s.t. π' is a prefix of π_i and π_i is a strict prefix of π.
- For every (π_i, A_i, λ_i), *s.t.* which π is a prefix of π_i write π_i as $\pi \cdot \pi'_i$, and replace it by $\pi' \cdot \pi'_i$.

It can be checked that the resulting S' is shorter, because there is at least one (π_i, A_i, λ_i) with $\pi_i = \pi'$ which will be removed. Moreover, S' still represents an open branch of the tableaux procedure without blocking. Hence we can continue the procedure and obtain an open branch on which the blocking rule is not applicable. □

4. Resolving Overloads in Types and Adjectives

Until now we have insisted that field and adjective names are always unique. In Example 1.10, we used sub, sub_1, sub_2, and sub_n as different variations of the name sub, dependent on whether we were taking a subformula of a negated formula, a formula constructed by a binary operator, or a formula constructed by an n-ary operator.

Similarly, we did not consider reuse of adjective names between different types. In reality, it is perfectly possible to have different types of formulas, for example modal, first-order and propositional, and define different nnf adjectives on each of them.

Modern programming languages like C^{++} or Java allow the use of ambiguous names which are made unambiguous by the compiler. For example, in C^{++}, one can define different print operators << on different types, and the compiler will pick the right one, when the programmer writes <<. This is called *overload resolution*. Without overload resolution, the programmer needs to invent a new name for every type that needs to be printed. For example, in C one has to include the type in the name of a print function, like `printfol` or `printmodal`.

We want overload resolution in our programming language: it should be possible that the user defines different nnf adjectives on different types and reuses the same name 'nnf' for them. Similarly, we want that the user can call all variations sub, sub_1, sub_2 and sub_n, just 'sub'.

Concretely, we want overload resolution on field names, adjective names, and function names. There will be no overload on type names, because we think it is unfeasable, and allowing ambiguous type names would result in ambiguous code.

In order to handle overload resolution, we introduce what we call *inexact identifiers*. Inexact identifiers are the identifiers that are used by the programmer in the program. We will call the identifiers that we have been using until now, *exact identifiers*. We assume a function $v^?$ that maps exact identifiers v to their inexact representations. The inexact representations are used in the program. As an example, one can introduce an inexact name sub and set $\mathrm{sub}^? = \mathrm{sub}^?_1 = \mathrm{sub}^?_2 = \mathrm{sub}^?_n = \mathrm{sub}$. Whenever identifier 'sub' occurs in the program, the compiler has to find out which of the exact variations of sub is meant. We define inexact identifiers:

Definition 4.1. *We assume an infinite set of inexact identifiers. We assume a map $v^?$ that maps exact identifiers v to inexact identifiers.*

Now we explain how inexact identifiers are used in the program. In order to do that, we modify the definitions of type and of adjective. In Definition 1.2, in the definition of compound types, we make the following modifications:

1. the identifiers $v_1, \ldots, v_n$, are inexact.

2. the identifier v is inexact.
3. identifier s is inexact.

In Definition 1.5, we will allow the identifiers v and f to be inexact. Apart from that, there are no changes.

Example 4.2. *We define propositional and multimodal logic:*

$$\text{ident} \;:=\; (\,\text{c}\colon \text{char}\,)^*.$$

$$\text{prop} \;:=\; \text{op}\colon \text{selector},\ \text{op}? \begin{pmatrix} ?\text{var} & \Rightarrow & \text{ident}, (\,)^* \\ ?\text{not} & \Rightarrow & \text{sub}\colon \text{prop},\ (\,)^* \\ ?\text{implies}\vee ?\text{equiv} & \Rightarrow & \text{sub}_1\colon \text{prop},\ \text{sub}_2\colon \text{prop},\ (\,)^* \\ ?\text{and}\vee ?\text{or} & \Rightarrow & (\,\text{sub}\colon \text{prop}\,)^* \end{pmatrix}$$

Similarly, one can define multimodal logic with inexact identifiers:

$$\text{modal} \;:=\; \text{op}\colon \text{selector},\ \text{op}? \begin{pmatrix} ?\text{var} & \Rightarrow & \text{ident}, (\,)^* \\ ?\text{not} & \Rightarrow & \text{sub}\colon \text{prop},\ (\,)^* \\ ?\text{implies}\vee ?\text{equiv} & \Rightarrow & \text{sub}_1\colon \text{prop},\ \text{sub}_2\colon \text{prop},\ (\,)^* \\ ?\text{and}, ?\text{or} & \Rightarrow & (\,\text{sub}\colon \text{prop}\,)^* \\ ?\text{box}\vee ?\text{dia} & \Rightarrow & \text{sub}\colon \text{prop},\ (\,)^* \end{pmatrix}$$

In Example 4.2, the inexact field name sub occurs both in prop and in modal. In type prop, it occurs one time as scalar field, and one time as repeated field. If one knows that that $f.\text{op} = ?\text{var}$, one can access $f.\text{sub}$. If one knows that $f.\text{op} \in \{?\text{and}, ?\text{or}\}$, one can access $f.\text{sub}[i]$. Fieldname sub also occurs three times in type modal, two times as a scalar field, and one time as a repeated field.

Example 4.3. *The adjectives* atom *and* literal *can be defined both on* prop *and on* modal*:*

$$\begin{array}{lcl} \text{atom}\colon \text{prop} & := & \text{op}(?\text{var}) \\ \text{atom}\colon \text{modal} & := & \text{op}(?\text{var}) \\ \text{literal}\colon \text{prop} & := & \text{atom} \vee \text{op}(?\text{not}) \wedge \text{sub}(?\text{atom}) \\ \text{literal}\colon \text{modal} & := & \text{atom} \vee \text{op}(?\text{not}) \wedge \text{sub}(?\text{atom}) \end{array}$$

Similarly, NNF *can be defined both on* prop *and on* modal*:*

$$\text{nnf}\colon \text{prop} \;:=\; \bigvee \begin{cases} \text{literal} \\ \text{op}(?\text{and}\vee ?\text{or}) \wedge \text{sub}(\forall \text{nnf}) \end{cases}$$

$$\text{nnf}\colon \text{prop} \;:=\; \bigvee \begin{cases} \text{literal} \\ \text{op}(?\text{and}\vee ?\text{or}) \wedge \text{sub}(\forall \text{nnf}) \\ \text{op}(?\text{box}\vee ?\text{dia}) \wedge \text{sub}(\text{nnf}) \end{cases}$$

In the example, there are different occurrences of sub, and it has to be determined which of the possible definitions is being referred to. Before we start discussing the treatment of inexact identifiers, note that defined identifiers in Σ_S, Σ_C and Σ_A are always exact. Moreover, we do not allow the use of ambiguous type names. This means that when a defined or built-in type is used, it must be referred to by its exact name.

Resolving of inexact identifiers starts with a preprocessing step on Σ_S and Σ_C. During this step, declared fields are replaced by unique exact identifiers, and any adjectives used in field declarations are moved to Σ_A. This is done by replacing them with a unique exact identifier, and defining this identifier in Σ_A. After the preprocessing step, all inexact identifiers are in adjectives in Σ_A.

Adjectives occurring in the types of function declarations will be dealt with in the same way. They are replaced with identifiers, defined in Σ_A, somewhat similar to the way subformulas are replaced in the CNF transformation ([8]). In our case the goal is not efficiency, but to make overload resolution possible in the first place.

We allow reuse of the same inexact field name in a compound type, as long as the different occurrences of the field name are in different states of the type. For example in prop in Example 4.2, it is possible to reuse fieldname sub between formulas built by ?not and formulas built by ?and∨?or, but renaming both fields sub_1 and sub_2 into sub at the same time would be impossible. It would be still possible to rename one of them into sub.

We first discuss how type definitions in Σ_S and Σ_C are processed.

Definition 4.4. *Let $D = (\, w_1 \colon W_1, \ldots, w_n \colon W_n \,)$ be a sequence of declarations. Let $w_{n+1} \colon W_{n+1}$ be a single declaration. We define*

$$D + (w_{n+1} \colon W_{n+1}) \; = \; (\, w_1 \colon W_1, \ldots, w_{n+1} \colon W_{n+1} \,).$$

Definition 4.5. *We define procedure* **preprocsimple**(T) *that preprocesses a simple type T. The result is again a simple type. It makes additions to Σ_A in the process.*
The implementation of **preprocsimple**(T) *is as follows:*
- *Let $A = \mathrm{ADJ}(T)$, and let $T' = \mathrm{IMPL}(T)$. If $A = \top$, then return T'. If $A \neq \top$, then create a new exact identifier α, assign $\Sigma_A(\alpha) := (T', A)$, and return $T' \circ \alpha$.*

Definition 4.6. *We define procedure* **preproccompound**(C, D) *that preprocesses a compound type C within declaration context D. The second argument is is used for checking that no inexact identifier is reused on a possible realization of C.*
- *If C has form $(\, v_1 \colon V_1, \ldots, v_n \colon V_n \,)^*$, then*
  - *if there exist $(w \colon W) \in D$ and $1 \leq i \leq n$, s.t. $w^? = v_i$, then create an error.*
  - *if there exist $1 \leq i < j \leq n$, s.t. $v_i = v_j$, then create an error.*
  - *Otherwise, create new, exact identifiers e_i for each v_i, and set $e_i^? = v_i$.*
  - *Set $V_i' =$* **preprocsimple**(V_i).
  - *Return $(\, e_1 \colon V_1', \ldots, e_n \colon V_n' \,)^*$.*
- *If C has form $(v \colon V,\ C')$ then if there is a $(w \colon W) \in D$, s.t. $w^? = v$, create an error.*
  Otherwise, let e be a new, exact identifier for v. Set $e^? = v$. Set $V' =$ **preprocsimple**(V). *Return*

$$e \colon V',\ \textbf{preproccompound}(C', D + (e \colon V')\,).$$

- *If C has form $s?(A_1 \Rightarrow C_1, \ldots, A_m \Rightarrow C_m)$, then there must a be unique $(w \colon W) \in D$, s.t. $w^? = s$.*
  - *If no such w exists, or w is not unique, then create an error.*
  - *Otherwise, create new exact identifiers $\alpha_1, \ldots, \alpha_m$, and set $\Sigma_A(\alpha_j) = (W, A_j)$, for each $1 \leq j \leq m$.*
  - *After that, return*

$$w?(\, \alpha_1 \Rightarrow \textbf{preproccompound}(C_1, D), \ldots, \alpha_m \Rightarrow \textbf{preproccompound}(C_m, D)\,).$$

We now apply procedures **preprocsimple** and **preproccompound** as follows:
- For every identifier v in the domain of Σ_S, replace $\Sigma_S(v)$ by **preprocsimple**$(\,\Sigma_S(v)\,)$.
- For every identifier v in the domain of Σ_C, replace $\Sigma_C(v)$ by **preproccompound**$(\,\Sigma_C(v),\ (\,)\,)$.

Example 4.7. *After being replaced by* **preproccompound**, *the definition of* prop *will have the following form in Σ_C :*

$$\mathrm{prop} \; := \mathrm{op} \colon \mathrm{selector}, \; \left\{ \begin{array}{lcl} \alpha_1 & \Rightarrow & \mathrm{ident},\ (\,)^* \\ \alpha_2 & \Rightarrow & \mathrm{sub} \colon \mathrm{prop},\ (\,)^* \\ \alpha_3 & \Rightarrow & \mathrm{sub}_1 \colon \mathrm{prop},\ \mathrm{sub}_2 \colon \mathrm{prop},\ (\,)^* \\ \alpha_4 & \Rightarrow & (\, \mathrm{sub}_n \colon \mathrm{prop} \,)^* \end{array} \right.$$

$$\begin{aligned}
\alpha_1 &: \text{selector} := (\,\text{selector}, ?\text{var}\,)\\
\alpha_2 &: \text{selector} := (\,\text{selector}, ?\text{not}\,)\\
\alpha_3 &: \text{selector} := (\,\text{selector}, ?\text{implies}\vee ?\text{equiv}\,)\\
\alpha_4 &: \text{selector} := (\,\text{selector}, ?\text{and}\vee ?\text{or}\,)
\end{aligned}$$

At this moment, we have moved all inexactness into Σ_A. Adjective definitions in Σ_A have form $\Sigma_A(v) = (T, A)$, where v is an exact identifier, T is the type on which the defined adjective can be aplied, and A is the adjective. Unfortunately, T is a simple type, which also may contain inexact adjectives. As an example where this could occur, one could define adjective cnf (Example 1.10) on **prop**∘nnf instead of prop. The definition would have form $\Sigma_A(\text{cnf}) = (\,\textbf{prop}\circ\text{nnf},\ A'\,)$, where A' is an inexact version of the expression given in Example 1.10, and nnf has a definition of form $\Sigma_A(\text{nnf}) = (\,\text{prop}, A''\,)$, where A'' is the expression of Example 4.3.

In order to solve this, we apply a preprocessing stage on Σ_A, similar to the preprocessing of Σ_S. We replace all adjectives occuring in domains by exact identifiers, while adding the definitions to Σ_A.

- For every identifier v in the domain of Σ_A, write $\Sigma_A(v)$ as (T, A). After that, replace $\Sigma_A(v)$ by $(\,\textbf{preprocsimple}(T), A)$.

Note that the call of **preprocsimple** may implicitly add new identifiers to Σ_A. These new identifiers need not be considered because their domain is an implementation type without adjectives. Hence there is no risk of non-termination. Unfortunately, there potentially exists a circular dependency between preconditions of adjectives that is hard to detect, as illustrated by the following example:

Example 4.8. *Suppose that Σ_A contains adjective definitions with the following circular structure:*

$$\begin{aligned}
v_1 &:= (\,T_1\circ\alpha_1, \cdots)\\
v_2 &:= (\,T_2\circ\alpha_2, \cdots)\\
\cdots&\\
\alpha_1 &:= (\,T_1, \cdots\ v_2^?\ \cdots)\\
\alpha_2 &:= (\,T_2, \cdots\ v_1^?\ \cdots)
\end{aligned}$$

If at some other point an occurrence of $v_1^?$ needs to be resolved, then v_1 is an overload candidate. In order to decide if v_1 should be used, one has to resolve the adjective of $\Sigma_A(\alpha_1)$, which contains $v_2^?$. In order to decide if v_2 is an overload candidate for this occurrence of $v_2^?$, one has to resolve the adjective in $\Sigma(\alpha_2)$, which again contains $v_1^?$. This circular dependency is not solvable. Note that using $v_1^?$ or $v_2^?$ inside the right hand side of v_1 or v_2 is unproblematic.

The circular dependency in Example 4.8 is unsolvable, so the compiler has to reject it. Unfortunately, it is difficult to detect, because it only exists if the use of $v_2^?$ in $\Sigma(\alpha_1)$ is applied on implementation type T_2, and the use of $v_1^?$ in $\Sigma(\alpha_2)$ is applied on implementation type T_1. If for example the use of $v_2^?$ in $\Sigma(\alpha_1)$ is not on T_2, then v_2 is not a candidate for overload, and there is no problem.

We solve this problem by starting to resolve an adjective definition, and whenever we encounter a precondition that has to be resolved first, we recursively try to solve this precondition. If this iteratively results in returning to the original definition, we reject Σ_A. In order to detect when we returned to the original definition, we maintain a set E of adjective definitions that we have encountered already. A circular dependency occurs when we need to resolve the precondition of an identifier that already occurs in E.

At this stage, all inexact identifiers are confined in Σ_A, in the second components A of the definitions $\Sigma_A(v) = (T, A)$. We give the procedure:

Definition 4.9. *Let Σ_A be the map of inexact adjective definitions. We define procedure* RESOLVE(v) *that tries to resolve the overloads in $\Sigma_A(v)$.*

It uses a set E of identifiers that were already encountered. Initially, $E = \emptyset$. The purpose of E is to detect circular dependencies of the type shown in Example 4.8. The implementation is as follows:

- *If v has no definition in $\Sigma_A(v)$, then the result is an error.*
- *If $v \in E$, then the result is also an error. This means that a circular dependency was detected.*
- *Otherwise, add v to E.*
- *Write $\Sigma_A(v)$ in the form (U, A). If U has form $(T \circ w)$, then call* RESOLVE(w).
- *Let $A' =$* RESOLVE$_T(\emptyset, A)$, *and replace $\Sigma_A(v)$ by A'. Note that this function calls* RESOLVE$_T$, *that will be defined in Definition 4.10.*

Function RESOLVE(v) possibly calls itself in order to make w exact. This will be detected, because we will have $v \in E$. There is no way to detect such circularity a priori, because its existence depends on the way overloads are resolved.

The second procedure RESOLVE$_T(\Gamma, A)$ tries to resolve the fields and adjectives in the non-exact adjective A in context Γ. Context Γ is needed because it may contain preconditions of fields.

In case more than one overload candidate exists, we will take the nearest fit. This approach is used by C^{++} and Java. For example, if some class A is a subclass of B, which is a subclass of C, and some function f has definitions $f(A)$ and $f(C)$, then the call $f(a)$ will be resolved as $f(A)$, while the call $f(b)$ will be resolved as $f(C)$. We have no notion of subclass, but we have implication between adjectives. Using implication between adjectives, the rule becomes as follows: If an application of some inexact identifier f has different overload candidates $f_1, \dots, f_n$, with $f_1^? = \cdots f_n^? = f$, s.t. each f_i is defined on adjective A_i, then we resolve f as follows: If there is a unique A_i, s.t. A_i implies all of $A_1, \dots, A_n$, then f will be resolved as f_i. In the definition, we will write $\Gamma \models A$, which means that $\Gamma, \neg A$ is unsatisfiable. It should be noted that this is always in the context of a fixed Σ_S, Σ_C, and Σ_A. We do not want to use the notation $\Sigma_S, \Sigma_C, \Sigma_A, \Gamma \models A$, because it becomes too long.

Definition 4.10. *Let T be an implementation type, let Γ be a set of exact adjectives applicable on T, and let A be an inexact adjective.* RESOLVE$_T(\Gamma, A)$ *tries to resolve the overloads in A when applied on T in context Γ. If it succeeds it returns the exact version of A. We define* RESOLVE$_T(\Gamma, A)$ *by cases on the form of A.*

- *If A is an (inexact) identifier v, then let $v_1, \dots, v_n$ be the adjectives defined in Σ_A, that have $v_i^? = v$ and for which $\Sigma_A(v_i)$ has form (T, A_i) or $(T \circ w_i, A_i)$.*
  Set $C = \emptyset$. For each $i \in \{1, \dots, n\}$, do the following:
  - *If w_i is absent, add i to C.*
  - *Otherwise, call* RESOLVE(w_i). *(This is the first version, defined in Definition 4.9.) If after that, $\Gamma \models w_i$, then add i to C.*

  At this moment C is a subset of $\{1, \dots, n\}$ containing the candidates that can still be considered as overloads for v. For $i \in C$ do:
  - *for $j \in C \backslash \{i\}$ do:*
    - *If $\Gamma, w_i \models w_j$, then remove j from C.*

  If $\|C\| \neq 1$, then create an error message. Otherwise, return v_i where i is the unique element of C.
- *If A is a constant c, then if c has primitive type T, return c. Otherwise create an error.*
- *If A has form $c^{\geq}$, and c has a primitive type that is not* selector, *then return $c^{\geq}$. Otherwise, create an error.*
- *If A has form $f(A')$, then let $f_1, \dots, f_n$ be the fields (scalar or repeated) defined on type T, that have $f_i^? = f$. Note that T must be a defined compound type, because primitive types have no fields. Set $C = \{1, \dots, n\}$. For each $i \in C$ do:*
  - *let* PREC$(f_i) = g_{i,1}(A_{i,1}) \wedge \cdots \wedge g_{i,k_i}(A_{i,k_i})$ *with $k_i \geq 0$, be the precondition of field f_i as defined in Definition 1.12. Each $A_{i,j}$ is either a implementation type or has form $T_{i,j} \circ w_{i,j}$ with $w_{i,j}$ an exact identifier defined on implementation type $T_{i,j}$. For every $w_{i,j}$ $(1 \leq j \leq k_i)$ that is present, call* RESOLVE$(w_{i,j})$, *defined in Definition 4.9.*
  - *After that, check that $\Gamma \models g_{i,1}(A_{i,1}) \wedge \cdots \wedge g_{i,k_n}(A_{i,k_i})$. If not, then remove i from C.*

If $\| C \| \neq 1$, then the result is error. Otherwise, let i be the unique element of C. Assume that the declaration of f_i has form $f_i \colon W$. If f_i is a scalar field, then return

$$f_i(\,\text{RESOLVE}_{\text{IMPL}(W)}(\text{ADJ}(W), A')\,).$$

if f_i is a repeated field, then return

$$f_i(\,\text{RESOLVE}^{*}_{\text{IMPL}(W)}(\text{ADJ}(W), A')\,).$$

In the latter case, we called RESOLVE^{*} *defined below.*

- *If A has form $A_1 \vee \cdots \vee A_n$, then*

$$\text{RESOLVE}_T(\Gamma, A) = \text{RESOLVE}_T(\Gamma, A_1) \vee \cdots \vee \text{RESOLVE}_T(\Gamma, A_n).$$

- *If A has form $A_1 \wedge \cdots \wedge A_n$, then set $\Gamma_1 = \Gamma$. For $i = 1$ to n do the following:*
  - *Let $A'_i = \text{RESOLVE}_T(\Gamma_i, A_i)$.*
  - *Set $\Gamma_{i+1} = \Gamma_i \cup \{A'_i\}$.*

  After that, return $A'_1 \wedge \cdots \wedge A'_n$.

Next we define RESOLVE^{*} *which resolves repeated fields.*

- *If A has form $\forall A'$, then return $\forall\text{RESOLVE}_T(\emptyset, A')$.*
- *If A has form $\exists A'$, then return $\exists\text{RESOLVE}_T(\emptyset, A')$.*
- *If $A =$ **empty**, and T is not a compound type, then the result is an error. Otherwise return* **empty**.

In order to make nnf of Example 4.3 exact, one has to start by calling RESOLVE(nnf). Procedure RESOLVE will insert nnf into E, and call $\text{RESOLVE}_{\text{prop}}(\emptyset, A)$, with A the expanded definition of nnf. Since A is a disjunction, $\text{RESOLVE}_{\text{prop}}(\emptyset, A)$ will process the disjuncts independently.

The first disjunct equals literal. Procedure $\text{RESOLVE}_{\text{prop}}(\emptyset, \text{literal})$ will establish that literal is the unique overload and return literal. It will not look at the definition of literal. If one wants to resolve the overloads in the definition of literal, one must call RESOLVE(literal) separately.

The second disjunct equals $\text{op}(\text{?and}\vee\text{?or}) \wedge \text{sub}(\forall\text{nnf})$. $\text{RESOLVE}_{\text{prop}}(\,\emptyset, \text{op}(\text{?and}\vee\text{?or}) \wedge \text{sub}(\forall\text{nnf})\,)$ will recursively call $\text{RESOLVE}_{\text{prop}}(\,\text{op}(\text{?and}\vee\text{?or})\,)$, which will resolve op (inexact) into op (exact) and call $\text{RESOLVE}_{\text{selector}}(\text{?and}\vee\text{?or})$, which will return $\text{?and}\vee\text{?or}$ since both are constants. After that, it will call

$$\text{RESOLVE}_{\text{prop}}(\,\text{op}(\,\text{?and}\vee\text{?or}),\ \text{sub}(\forall\text{nnf})\,).$$

There are two possible overloads for sub, namely sub for the ?not case, and $\text{sub}_{\forall}$. We have $\text{PREC(sub)} = \text{op(?not)}$, and $\text{PREC}(\text{sub}_{\forall}) = \text{op}(\text{?and}\vee\text{?or})$. Since only the latter is provable from the premiss, $\text{sub}_{\forall}$ will be picked. Since $\text{sub}_{\forall}$ is a repeated field, the procedure will recursively call $\text{RESOLVE}^{*}_{\text{prop}}(\emptyset, \forall\text{nnf})$, which will recursively call $\text{RESOLVE}_{\text{prop}}(\emptyset, \text{nnf})$. This call will return nnf without expanding it, after which the original call of $\text{RESOLVE}_{\text{prop}}$ will construct the complete exact overload

$$\bigvee \begin{cases} \text{literal} \\ \text{op}(\text{?and}\vee\text{?or}) \vee \text{sub}(\forall\text{nnf}) \end{cases}$$

5. Conclusions

Our goal is to develop and implement an efficient programming language in which it is convenient to implement algorithms on trees whose forms are very different.

In order to obtain this, we have defined a flexible type system together with a way of refining these types by means of adjectives. The adjectives are intended as a replacement for matching in functional languages. In order to make this replacement possible, we have given a precise semantics for adjectives, so that adjectives can be evaluated on concrete data.

We provided a terminating tableaux calculus for deciding propositional relations between adjectives, and applied this procedure to overload resolution in imprecise formulations of adjectives. The overload resolution procedure replaces ambiguous field references in adjective definitions by exact field references. A similar algorithm can be used for resolving ambiguous overloads in function calls.

6. Acknowledgements

This work gained from discussions with Cláudia Nalon. We thank Nazarbayev University for supporting this research through the Faculty Development Competitive Research Grant Program (FDCRGP) number 021220FD1651.

Список литературы

[1] Y. Minsky, A. Madhavapeddy и J. Hickey, *Real World OCaml (functional programming for the masses).* O'Reilly, 2013, ISBN: 978-1-449-32391-2.

[2] G. Hutton, *Programming in Haskell (second edition).* Cambridge University Press, 2016, ISBN: 978-1-316-62622-1.

[3] M. Odersky, L. Spoon и B. Venners, *Programming in Scala Third Edition.* Artima Press, 2007-2016, ISBN: 0-9815316-8-7.

[4] P. Rondon, M. Kawaguchi и R. Jhala, "Liquid Types", в *Programming Language Design and Implementation (PLDI)*, R. Gupta и S. Amarasinghe, ред., сер. ACM, ACM, 2009, с. 159—169.

[5] R. Jhala и N. Vazou, "Refinement Types: A Tutorial", *Foundations and Trends in Programming Languages*, т. 6, № 3–4, с. 159—317, 2021, ISSN: 2325-1107. DOI: 10.1561/2500000032. url: http://dx.doi.org/10.1561/2500000032.

[6] S. Klabnik и C. Nichols, *The Rust Programming Language.* No Starch Press, 2019, ISBN: 978-1718500440.

[7] Michael Gelfond и Vladimar Lifschitz, "The Stable Model Semantics for Logic Programming", в *Fifth International Conference and Symposium on Logic Programming*, R. Kowalski и K. Bowen, ред., MIT Press, 1988, с. 1070—1080.

[8] G. Tseytin, "On the complexity of Derivation in the Propositional Calculus", в *Studies in Constructive Mathematics and Mathematical Logic, Part II*, A. Slisenko, ред., Zapiski Nauchnykh Seminarov, 1970, с. 115—125.

References

[1] Y. Minsky, A. Madhavapeddy, and J. Hickey, *Real World OCaml (functional programming for the masses).* O'Reilly, 2013, ISBN: 978-1-449-32391-2.

[2] G. Hutton, *Programming in Haskell (second edition).* Cambridge University Press, 2016, ISBN: 978-1-316-62622-1.

[3] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala Third Edition.* Artima Press, 2007-2016, ISBN: 0-9815316-8-7.

[4] P. Rondon, M. Kawaguchi, and R. Jhala, "Liquid Types", in *Programming Language Design and Implementation (PLDI)*, R. Gupta and S. Amarasinghe, Eds., ser. ACM, ACM, 2009, pp. 159–169.

[5] R. Jhala and N. Vazou, "Refinement Types: A Tutorial", *Foundations and Trends in Programming Languages*, vol. 6, no. 3–4, pp. 159–317, 2021, ISSN: 2325-1107. DOI: 10.1561/2500000032. [Online]. Available: http://dx.doi.org/10.1561/2500000032.

[6] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, 2019, ISBN: 978-1718500440.

[7] Michael Gelfond and Vladimar Lifschitz, “The Stable Model Semantics for Logic Programming”, in *Fifth International Conference and Symposium on Logic Programming*, R. Kowalski and K. Bowen, Eds., MIT Press, 1988, pp. 1070–1080.

[8] G. Tseytin, “On the complexity of Derivation in the Propositional Calculus”, in *Studies in Constructive Mathematics and Mathematical Logic, Part II*, A. Slisenko, Ed., Zapiski Nauchnykh Seminarov, 1970, pp. 115–125.