A Type System for Logical Formulas

Hans de Nivelle

School of Engineering and Digital Sciences, Nazarbayev University, Nur-Sultan City, Kazakhstan, hans.denivelle@nu.edu.kz

Abstract. We want to develop a programming language that is optimized for the implementation of logical formulas and logical algorithms. We want this language to be statically typed. This introduces some challenges to the type system: Logical formulas are tree-like structures that are recursively defined by different subcases that can have very different forms.
We present a type system in which such types can be concisely defined. In order to determine with which subcase we are currently dealing, we use subtypes (which we call 'adjectives'). We show that the type and adjective system (membership and inclusion) are decidable by means of translation into regular tree automata.

1 Introduction

Our goal is to develop a programming language that is specialized for the implementation of algorithms working on logical proofs and formulas. Before, we have implemented theorem provers in C^{++}. (See [4] or [5]). Such theorem provers typically consist of two parts: The first part transforms the logic of interest into a normal form. This normal form is much simpler than the logic of interest. Proof search takes place on the normal form. For this part, C^{++} is a suitable language. Speed is important, and the data format is simple. However, if one needs more operations on the logic of interest, for example simplifications, translations between different logics, or proof transformations, then C^{++} is not suitable. We have considered using alternative programming languages, like Scala ([9]), OCaml ([7]) or Haskell ([6]), but none of these languages is suitable for our goals. We need a language in which one can represent state (which implies that the languages must have some imperative features), that runs natively with a good interface to C^{++}, while still allowing for high level programming. We will develop a type system that is high level on one side, but which still has an efficient mapping into low level implementation on the other side. In our algorithms, we want to use *C* style switch instead of matching as main control mechanism, because it is more efficient. In order to make this possible, one needs dependent types, which means that the types of later fields can depend on the values of earlier fields. This corresponds to the fact that in a logical formula, existence of substructures depends on the logical operator. In order to remove the need for casts, we use adaptive typing. After every conditional statement, the compiler knows that the check succeeded, which limits the formulas that are possible at

this point. For example, if one checks that a formula has a binary operator, and the check succeeds, then one can be sure that the two subformulas exist, and one can safely access them. Once the need for type casts has been removed, it is possible to write code that is as concise as code that uses matching, but more efficient, because all existence checks are performed at compile time and C-style switch is cheap at run time. In order to be able to make the checks at compile time, one must be able to check inclusion relations between subtypes of formulas. Concretely, one must be able to check that the cases in a switch are exhaustive, (check that $\top \subseteq C_1 \vee \cdots \vee C_n$), check that the cases are non-overlapping (check that $C_i, C_j \subseteq \bot$ for $i \neq j$), and check that the current case implies the existence conditions of the subfields (check that $C_i \subseteq E$.) In this paper, we will develop the techniques for performing these checks. The outline of this paper is as follows: We will introduce the type system and explain its use below. In Section 2, we explain how types are translated into regular tree expressions. In Section 3, we translate the regular expressions into tree automata and show how to decide dependencies between them.

Definition 1. *We use primitive types* void, bool, char, index, integer, double, *and* selector. *There are no values of type* void. *The other types allow values. The types* bool, char, index, integer, double *are* ordered, *type* selector *is unordered.*

An index is an integer ≥ 0 whose size is big enough to index the biggest array that fits in memory. It is the equivalent of `size_t` in C^{++}. Type integer consists of integers of unbounded precision. Type selector contains only named constants. Its main purpose is to represent logical operators as named constants. Type selector is similar to enumeration types, but it has no computation, no order, cannot be converted to integer types, and there exists only one global selector type. There is no need to define different enumeration types, because every application automatically generates a subtype. Internally, selector is mapped into small integer contants. We write values of type selector as identifiers preceded by a question mark, e.g. `?and,?or,?implies,?equiv`, or `?zero,?succ`.

We will now define types and adjectives. Adjectives are similar to subtypes. We prefer the term 'adjective' over 'subtype', because adjectives overlap, are frequently created for one time use, and are not indended to hold independent objects. Types may contain adjectives in their definition, but we first define the types. Types can be either simple or compound. Compound types are simple structs, variant structs, or arrays.

Definition 2. *We first define the* simple types*:*

- *A primitive type is a simple type.*
- *An identifier is a simple type.*
- *If T is a type, A is an adjective, then $T \circ A$ is a simple type.*

Compound types are recursively defined as follows:

- struct(v_1:$V_1, \ldots, v_n$:V_n) *with $n \geq 0$ is a compound type.*

– array($v_1{:}V_1, \ldots, v_n{:}V_n;\ w_1{:}W_1, \ldots, w_m{:}W_m$) *is a compound type, where* $v_1{:}V_1, \ldots, v_n{:}V_n$ $(n \geq 0)$ *are the* fixed fields *of the array, and* $w_1{:}W_1, \ldots, w_m{:}W_m$ *with* $m > 0$ *are the* repeated fields *of the array.*
– variant($v_1{:}V_1, \ldots, v_n{:}V_n,\ s{:}S;\ \ A_1 \Rightarrow C_1, \ldots, A_m \Rightarrow C_m$) *is a compound type. Here* $v_1{:}V_1, \ldots, v_n{:}V_n$ $(n \geq 0)$ *are fixed fields,* $s{:}S$ *is the* pivot field*,* $A_1, \ldots, A_m$ $(m \geq 2)$ *are adjectives, and* $C_1, \ldots, C_m$ *are compound types again. The type of the pivot field* S *must be primitive.*

In all three cases, $V_1, \ldots, V_n,\ W_1, \ldots, W_m$ *must be simple types. In a* variant *type,* $C_1, \ldots, C_m$ *are compound types again.*

We distinguish between simple and compound types, because we want that every compound type has a name, which solves some technical problems. It has no consequences for expressivity, because one can always introduce a name for a compound type, and use the name as simple type.

Compound types are designed to have an efficient representation in C. In a compound type, all $V_1, \ldots, V_n,\ W_1, \ldots, W_m,\ S$ that are primitiv, are stored in place. The $V_1, \ldots, V_n,\ W_1, \ldots, W_m,\ S$ that have a compound type, are represented by a pointer to the heap. An array is like a C^{++} `std::vector`, it grows on demand, but is more flexible, because it can have a fixed prefix which is stored right before the repeated part in the same memory segment. As far as we know, C^{++} has no data structure that can do this. The elements in the fixed part are addressed just by their field names v_i, the elements in the repeated part are addressed by their field name $w_i(t)$, where t is of type index. We will not further discuss the intended low level implementation in this paper, and instead use a simple representation based on Prolog-style lists.

Definition 3. Adjectives *are recursively defined as follows:*

– *If* v *is an identifier (that should be defined as adjective), then* v *is an adjective.*
– *If* $c_1, \ldots, c_n$ $(n \geq 0)$ *are constants with a common primitive type, then* $\{c_1, \ldots, c_n\}$ *and its complement* $\{c_1, \ldots, c_n\}^C$ *are adjectives.*
– *If* c *is a constant of a primitive type, then* $(\leq c)$, $(\geq c)$, $(< c)$ *and* $> c)$ *are adjectives.*
– *If* A *is an adjective,* f *is an identifier (that should represent a fixed field), then* $f[A]$ *is an adjective.*
– *If* A *is an adjective,* $c_1, \ldots, c_n$ $(n > 0)$ *are constants of type* index*, and* f *is an identifier (that should represent a repeated field), then* $\{c_1, \ldots, c_n\}f[A]$ *is an adjective.*
– *If* A *is an adjective,* c *is a constant of type* index*, and* f *is an identifier (that should represent a repeated field), then* $(\forall^{\geq} c)f[A]$ *is an adjective. In case* $c = 1$*, we just write* $\forall f[A]$.
– *If* $c_1, \ldots, c_n$ $(n > 0)$ *are constants of type* index*, then* size($\{c_1, \ldots, c_n\}$) *is an adjective.*
– *If* c *is a constant of type* index*, then* size$(\geq c)$ *is an adjective.*
– *If* $A_1, \ldots, A_n$ *are adjectives, then* $A_1 \cap \cdots \cap A_n$ *and* $A_1 \cup \cdots \cup A_n$ *are adjectives. We write* $\top = \bigcap \emptyset$*, and* $\bot = \bigcup \emptyset$.

Adjectives are more subtle than they seem. They contain identifiers that need to be type checked and resolved before they can be used. In addition, the semantics of $\cap$ and $\cup$ is nonstandard.

The intuitive meaning of $\{c_1, \ldots, c_n\}$, $\{c_1, \ldots, c_n\}^C$, $(\leq c)$, $(\geq c)$, $(< c)$, and $(> c)$ is evident. For example, uindex $\circ$ (> 4) denotes an index greater than 4, and selector $\circ$ $\{?\text{true}, ?\text{false}\}$ denotes a selector that equals ?true or false.

The intuitive meaning of $f[A]$ is: whose f-field satisfies A. For example, if field f has type selector, then $f[\ \{?\text{and}, ?\text{or}\}\ ]$ means that f must be in $\{?\text{and}, ?\text{or}\}$. In order to be meaningful, the adjective has to be type checked. It has to be determined on which type $f[\ \{?\text{and}, ?\text{or}\ \}\ ]$ is being applied, and that this type has a field with name f whose type is selector. Fields can have preconditions corresponding to different options of variant, which have to be checked additionally.

The intuitive meaning of adjective $\{c_1, \ldots, c_n\}f[A]$ is: We are an array, and our $f(c_1), \ldots, f(c_n)$ satisfy A. The intuitive meaning of $\forall^{\geq c} f[A]$ is: We are an array, for which $f(i)$ satisfies A, when $i \geq c$. As for $f[A]$, adjectives of these two forms have to be type checked: It must be checked that the f field exists, is a repeated field, and A can be applied on its type. In addition, adjectives of form $\{c_1, \ldots, c_n\}f[A]$ should be used with care. If specific indices have specific properties, one should consider using struct and giving them names, instead of using an array. We defined this form mostly because it is possible in our proof system, not because we think it is useful. On the other hand, adjectives of form $\forall f[A]$ have many natural examples.

The intuitive meaning of size$(\{c_1, \ldots, c_n\})$ is: We are an array, whose size is in $\{c_1, \ldots, c_n\}$. The $c_1, \ldots, c_n$ must be constants. There is no way of expressing size in terms of a variable, for example stating .size() $\leq i$. The same remark as for $\{c_1, \ldots, c_n\}f[A]$ applies also here: If the size of the array is restricted, one should consider using variant or struct. The intuitive meaning of size$(\geq c)$ is: We are an array of size at least c.

The meaning of $A_1 \cap \cdots \cap A_n$ is: Fulfills all A_i. The meaning of $A_1 \cup \cdots \cup A_n$ is: Fulfills at least one A_i. Note that $\cup$ and $\cap$ are not complementary. The meaning of $\cap$ is closer to `&&`, while the meaning of $\cup$ is just logical disjunction.

We give an example that illustrates the intended application. We define propositional logic, where variables are represented by arrays of characters, the operations $\rightarrow$ and $\leftrightarrow$ have fixed arity 2, and $\wedge, \vee$ have variable arity. There is no need to define $\top$ or $\bot$ separately, because $\top = \bigwedge \emptyset$, and $\bot = \bigvee \emptyset$.

Example 1. We define propositional logic:

$$
\begin{array}{l}
\text{TYPE prop} := \text{variant(}\ \text{sel:selector;} \\
\quad \left\{ \begin{array}{ll}
\{?\text{var}\} & \Rightarrow \text{array(}\ ;c\text{:char}\ \text{)} \\
\{?\text{not}\} & \Rightarrow \text{struct(}\ \text{body:prop}\ \text{)} \\
\{?\text{implies}, ?\text{equiv}\} & \Rightarrow \text{struct(}\ \text{left:prop, right:prop}\ \text{)} \\
\{?\text{and}, ?\text{or}\} & \Rightarrow \text{array(}\ ;\text{sub:prop}\ \text{)}\ \text{)}
\end{array} \right.
\end{array}
$$

Negation normal form (NNF) can be defined by the following adjective:

$$\text{ADJ}\quad \text{nnf:prop} := \bigcup \begin{cases} \text{sel[\{?var\}]} \\ \text{sel[\{?not\}]} \cap \text{body[sel[\{?var\}]]} \\ \text{sel[\{?and, ?or\}]} \cap \forall\text{sub[nnf]} \end{cases}$$

Our long term goal is to be able to implement logic in a nice and efficient way. If f has type prop, then we want to be able to write code of the following form:

Example 2. We present a procedure that prints a logical formula:

```
void print( prop f ) :
switch f.sel :
    case ?var :
        for uindex i := 1 to f.size do :
            print( f.c(i) )
    case ?not : print( "( not " ); print( f.body ); print( " )" )
    case ?implies : print( "( "); print( f.left );
                    print( " implies "); print( f.right ); print(" )")
    case ?equiv : print( "( " ); print( f.left ); print( " equiv ")
                    print( f.right ); print( " ) " );
    case {?and, ?or} :
        if f.op = ?and && f.size = 0 then print( "true" ); return
        if f.op = ?or && f.size = 0 then print( "false" ); return
        if f.op = ?and then print( "and( " ); else print( "or( " )
        print( f.sub(1) )
        for uindex i := 2 to f.size do :
            print( ", " ); print( f.sub(i) ))
```

In order to do print a formula this, one needs to switch on f.sel. In each case of the switch, the compiler knows that it is dealing with an f of type f.sel[S], for $S \in \{?\text{var}, ?\text{not}, \text{implies}, \text{equiv}, ?\text{and}, ?\text{or}\}$. Therefore, it knows that the accessed subfields exist. Field size is a field that is defined for every option that has repeated fields. Access of repeated fields has form .sub(i), where sub is the field name and i must have type index. Note that, in this example, indices run from 1 to the size.

If we would be using C^{++} we would have to cast f into a subtype in every case of the switch, or use forms of subfield access that cannot be checked at compile time. In languages with matching, one would use matching instead of switch. It is less efficient, and less flexible. Note that in principle, adjectives of any form can be used in **switch** statements.

In order to derive the adjectives that apply to f at a given point in function print, we use *abstract interpretation*. (See [8] or [3]). The technique has been worked out, and will be elswehere.

Since field names, constructor names, function and adjective names can be overloaded, we need a framework for doing this. We do this by distinguishing between *inexact* and *exact* identifiers. The program contains inexact identifiers

which are overloaded by exact identifiers. During compilation, the compiler replaces the inexact identifiers by exact identifiers. One inexact identifier can have any number of exact instances. We will now introduce the *symbol table* which is the data structure that holds all type, adjective and field definitions. Names that are introduced in the symbol table are always exact.

Definition 4. *The* symbol table Σ *is a finite sequence of definitions of the following forms. The defined identifier is always exact.*

- *A* type definition *has form* TYPE $v := T$, *where* v *is an identifier, and* T *a type or the expansion set of a compound type.*
- *An* adjective definition *has form* ADJ $v{:}T := A$, *with* v *an identifier,* T *a type, and* A *an adjective.*
- *A* function definition *has form* FUNC $f{:}U(U_1, \ldots, U_a) := G$, *in which* f *is an identifier,* U *is the return type,* $U_1, \ldots, U_a \neq$ void *are the types of the arguments, and* G *a flow graph. We call* $a \geq 0$ *the* arity *of the function. Arguments types* $U_1, \ldots, U_a$ *can be marked as* reference arguments
- *A* fixed field definition *has form*

$$\text{FIELD } f{:}U(U_1) := c/\{\ \text{void}(A_{1,1}, A_{1,2}), \ldots, \text{void}(A_{m,1}, A_{m,2})\ \}.$$

  where f *is an identifier (name of the field),* U *is the type of the field,* U_1 *is the compound type of which* f *is a field,* p *is the relative position of the field, and each* $\text{void}(A_{i,1}, A_{i,2})$ *is a possible type of assignment to* f. *We have* $m \geq 2$ *iff* f *is a pivot field.*
- *A* repeated field definition *has form* FIELD$^{[]}$ $f{:}U(U_1, \text{index}) := (c, d)$, *where* f *is an identifier (name of the field),* U *is the type of the field,* U_1 *is the compound type of which* f *is a field,* c *is the number of fixed fields of the compound type, and* d *is the relative position of* f *among the repeated fields.*

Field definitions will be generated automatically from compound type definitions. In every array, we will call the first repeated field the *base field*, and we assume that it is an overload of the inexact identifier *repbase*, in addition to its declared name. In prop, the fields c and sub are base fields. Before we explain how field and constructor definitions are obtained, we give another example:

Example 3. Natural numbers can be defined as follows:

$$\text{TYPE nat} := \text{variant}(\ \text{sel:selector} \circ \{?\text{zero}, ?\text{succ}\};\ \begin{array}{l} ?\text{zero} \Rightarrow \text{struct}(\ ), \\ ?\text{succ} \Rightarrow \text{struct}(\ \text{pred:nat}\ )\ ).\end{array}$$

A natural number is either zero, or a successor. If we are zero, we have no further fields. If we are a successor, then we have one field (called pred), that is also a natural number. One can define adjectives iszero or issucc as follows:

$$\begin{array}{l} \text{ADJ iszero:nat} := \text{sel}[\ \{?\text{zero}\}\ ], \\ \text{ADJ issucc:nat} := \text{sel}[\ \{?\text{succ}\}\ ], \\ \text{ADJ even:nat} := \text{iszero} \cup (\ \text{issucc} \cap \text{pred}[\ \text{odd}\ ]\ ) \\ \text{ADJ odd:nat} := \text{issucc} \cap \text{pred}[\ \text{even}\ ] \end{array}$$

Note that we do not think that natural numbers should be implemented like this in real code. Example 3 is just an illustration of our type system. We will now explain how field definitions and constructors are obtained from compound types. This is surprisingly tricky.

Definition 5. *Let $(x_1, \ldots, x_n)$ be a sequence. We write $\overline{x}$ for the complete sequence. We write $(\cdot)$ for sequence concatenation, i.e. if $\overline{x} = (x_1, \ldots, x_n)$, $\overline{y} = (y_1, \ldots, y_m)$, then $\overline{x} \cdot \overline{y} = (x_1, \ldots, x_n,\ y_1, \ldots, y_m)$. We write ϵ for the empty sequence, and use power notation for repeated concatenation, i.e. $\overline{x}^0 = \epsilon$, $\overline{x}^{k+1} = \overline{x}^k \cdot \overline{x}$.*

We need to create field definitions and constructor definitions for every compound type. In case, a compound type has an array amongs its options, the set of constructors is infinite. Therefore, constructors have to be generated on the fly when they are needed. More precisely, when a constructor of arity a is called, the possible constructors of arity a have to be created and inserted into Σ so that they become available as overload. This can be done in time $O(a)$, since there is only one constructor per option for a given arity a.

In order to create the constructors and field definitions for a compouned type, we use *expansions*. An expansion of a compound type is obtained by making a choice for the variants that occur in it. The result is a 5-tuple $(\overline{v}, \overline{V}, \overline{A},\ \overline{w}, \overline{W})$, in which $\overline{v}$ are the field names, $\overline{V}$ are their types, and $\overline{A}$ are the adjectives that were collected on the way. If there is a repeated part, it is registered in $\overline{w}, \overline{W}$.

Definition 6. *Let C be a compound type. We define the expansion set $E(C)$ of C as $E(\epsilon, \epsilon, \epsilon,\ C)$, where $E(\overline{v}', \overline{V}', \overline{A}',\ C)$ is defined as follows:*

- *If C has form* struct$(\ v_1{:}V_1, \ldots, v_n{:}V_n\ )$, *then*

  $\{\ E(\overline{v}', \overline{w}', \overline{A}',\ C) = (\ \overline{v}' \cdot (v_1, \ldots, v_n),\ \ \overline{V}' \cdot (V_1, \ldots, V_n),\ \ \overline{A}' \cdot (\top)^n,\ \ \epsilon,\ \epsilon\ )\ \}.$

- *If C has form* array$(\ v_1{:}V_1, \ldots, v_n{:}V_n;\ w_1{:}W_1, \ldots, w_m{:}W_m\ )$, *then* $E(\overline{v}', \overline{w}', \overline{A}',\ C) =$

  $\{\ (\ \overline{v}' \cdot (v_1, \ldots, v_n),\ \overline{V}' \cdot (V_1, \ldots, V_n),\ \overline{A}' \cdot (\top)^n,\ (w_1, \ldots, w_m),\ (W_1, \ldots, W_m)\ )\ \}.$

- *If C has form*

  $$\text{variant}(\ v_1{:}V_1, \ldots, v_n{:}V_n,\ s{:}S;\quad A_1 \Rightarrow C_1, \ldots, A_m \Rightarrow C_m\ ),$$

  then $E(\overline{v}', \overline{w}', \overline{A}',\ C) =$

  $$\bigcup_{j=1}^{m} E(\overline{v}' \cdot (v_1, \ldots, v_n, s),\ \overline{V}' \cdot (V_1, \ldots, V_n, S),\ \overline{A}' \cdot (\top)^n \cdot (A_j),\ C_j).$$

For every $(\overline{v}, \overline{V}, \overline{A},\ \overline{w}, \overline{W}) \in E(C)$, we have $\|\overline{v}\| = \|\overline{V}\| = \|\overline{A}\|$, and $\|\overline{w}\| = \|\overline{W}\|$.

Function $E(\overline{v}', \overline{V}', \overline{A}',\ C)$ constructs the expansion set of C, while putting $(\overline{v}', \overline{V}', \overline{A}')$ in front of each expansion. Once we have the expansion set $E(C)$, we can completely discard C, because all information is now contained in $E(C)$. We will use $E(C)$ to generate the field definitions.

In principle, field definitions can be generated directly from C, but it is better to use $E(C)$, because then it is easier to merge fields that have a common initial sequence, which makes the use of such fields more flexible. For example, if C is

$$\text{variant}(\ s{:}\text{index} \circ \{1,2,3\};\ \begin{array}{l}\{1\} \Rightarrow \text{struct}(\ x{:}\text{double}, y{:}\text{double}, z{:}\text{double}\ ) \\ \{2\} \Rightarrow \text{struct}(\ x{:}\text{double}, y{:}\text{double}, i{:}\text{integer}\ ) \\ \{3\} \Rightarrow \text{struct}(\ x{:}\text{double}, b{:}\text{bool}\ )\ )\end{array}$$

then the three definitions of field x will end up in the same location in memory, and therefore can be defined as a single field which does not depend on the value of s. Similarly, the two occurrences of field y can be merged into a single field that is defined when $s \in \{1,2\}$. Because of this, we will define a field creation procedure that takes the expansion set as starting point. It iteratively assigns exact identifiers to the field declarations at position i in E, while simultaneously inserting the field definitions into Σ, and making sure that the same exact identifier is being assigned to fields that have the same inexact name, the same type, and all whose predecessors before it were assigned the same exact identifier.

Definition 7. *Let* TYPE $c := C$ *be a compound type definition, let* $E = E(C)$. *The field assignment procedure generates field definitions from* E, *and simultaneously replaces inexact field names in* E *by exact field names. Let* n *be the maximal length of a* $\overline{v}$, *for all* $(\overline{v}, \overline{V}, \overline{A}, \overline{w}, \overline{W}) \in E$. *For* $i = 1$ *to* n, *do the following:*

1. *As long as there exists a* $(\overline{v}, \overline{V}, \overline{A}, \overline{w}, \overline{W}) \in E$, *for which the field name* v_i *is not exact, let*

$$E' = \{\ (\overline{v}', \overline{V}', \overline{A}', \overline{w}', \overline{W}') \in E \mid \text{ for all } j \le i,\ v'_j = v_j \text{ and } V'_j = V_j\}.$$

   These are the expansions that agree with $(\overline{v}, \overline{V}, \overline{A}, \overline{w}, \overline{W})$ *up to and including position* i. *Note that all* v_j *with* $j < i$ *are exact, while* v_i *itself is inexact.*
2. *Create a fresh exact identifier* x *and replace all* v'_i *in* E *by* x *(and also in* E'*.)*
3. *Let* $Z = \bigcup (\ v_1[A'_1] \cap \cdots \cap v_{i-1}[A'_{i-1}]\ )$, *for which* $(\overline{v}', \overline{V}', \overline{A}', \overline{w}', \overline{W}') \in E'$. *Note that* $\bigcup$ *and* $\bigcap$ *are adjective operations, not set operations. It may be possible to simplify* Z *by removing the* $v_j[A'_j]$, *for which* $A'_j = \top$. *The identifiers* $v_1, \ldots, v_{i-1}$ *are exact.*
4. *For each* $e = (\overline{v}', \overline{V}', \overline{A}', \overline{w}', \overline{W}') \in E'$, *let*

$$Y_e = \text{void}(\ c \circ (\ v_1[A'_1] \cap \cdots \cap v_{i-1}[A'_{i-1}]\ ),\ V'_i \circ A'_i\ ).$$

   Append the field definition

$$\text{FIELD } x{:}V_i(c \circ Z) := \quad i/\{Y_e \mid e \in E'\}$$

   to Σ. *Again it is possible to simplify the* Y_e *by removing trivial* $v_j[\top]$.

For repeated fields, we need to do the same, but it is easier because there is no special treatment of common initial sequences for arrays. For every $(\overline{v}, \overline{V}, \overline{A}, \overline{w}, \overline{W}) \in E(C)$*, for every* $j \leq \|\overline{W}\|$*, do the following:*

1. *Create a fresh exact identifier* x*, and replace* w_j *by* x*.*
2. *Using* $n = \|\overline{V}\|$*, append*

$$\text{FIELD}^{[]}\ w_i{:}W_j(\ c \circ (v_1[A_1] \cap \cdots \cap v_n[A_n]\ ), \text{index}\ ) := \quad (\ \|\overline{V}\|,\ \|\overline{W}\|\ )$$

to Σ*.*

Definition 7 can generate quite ugly adjective expressions. It is complicated by the fact that we want to merge fields with a common initial sequence. In practical examples that we considered, there are no nested optional fields, which greatly simplifies the possible adjective expressions. We considered restricting definition 7 but writing out the restriction is as ugly as writing out the unrestricted version.

$$\left\{ \begin{array}{l} (\ (s,x,y,z),\ (\ \text{index} \circ \{1,2,3\}, \text{double}, \text{double}, \text{double}\ ), (\ \{1\}, \top, \top, \top\ ), \epsilon, \epsilon), \\ (\ (s,x,y,i),\ (\ \text{index} \circ \{1,2,3\}, \text{double}, \text{double}, \text{integer}\ ), (\ \{2\}, \top, \top, \top\ ), \epsilon, \epsilon), \\ (\ (s,x,b),\ (\ \text{index} \circ \{1,2,3\}, \text{double}, \text{bool}\ ), (\ \{3\}, \top, \top, \top\ ), \epsilon, \epsilon) \end{array} \right\}$$

Assignments to pivot fields must be restricted in such a way that they do not bring the type in an undefined state. This greatly complicates the definition of FIELD. We considered several approaches and none of them was prettier than the one provided in this paper.

We also need constructors. These must be generated as needed, because their number is infinite in the presence of repeated fields.

Example 4. We give the expansion set of the compound type nat :

$$\left\{ \begin{array}{l} (\ (\text{sel}), (?\text{selector} \circ \{?\text{zero}, ?\text{succ}\}), (\{?\text{zero}\}), \epsilon, \epsilon\ ), \\ (\ (\text{sel}, \text{pred}), (?\text{selector} \circ \{?\text{zero}, ?\text{succ}\}, \text{nat}), (\{?\text{succ}\}, \top), \epsilon, \epsilon\ ) \end{array} \right\}$$

Example 5. We give the expansion set of the compound type prop :

$$\left\{ \begin{array}{l} (\ (\text{sel}), (?\text{selector} \circ S), (\{?\text{var}\}), (c), (\text{char})\ ) \\ (\ (\text{sel}, \text{body}), (?\text{selector} \circ S, \text{prop}), (\{?\text{not}\}, \top), \epsilon, \epsilon\ ) \\ (\ (\text{sel}, \text{left}, \text{right}), (\text{selector} \circ S, \text{prop}, \text{prop}), (\{?\text{implies}, ?\text{equiv}\}, \top, \top), \epsilon, \epsilon\ ), \\ (\ (\text{sel}), (\text{selector} \circ S), (\{?\text{and}, ?\text{or}\}), (\text{sub}), (\text{prop})\ ) \end{array} \right\}$$

2 Regular Expressions over Trees

We want to obtain a decision procedure that decides inclusions between adjectives. In order to obtain this, we will first represent data by Prolog-style lists over the primitive types. After that, we translate adjectives and types into a kind of tree regular expressions. In the next section, we translate the expression into tree automata, and show how to decide type inclusions.

Definition 8. *We recursively define* terms*:*

- *Constants of the primitive types* bool, char, index, integer, double, *or* selector *are terms.*
- **nil** *is a term. If* t_1 *and* t_2 *are terms, then* $\mathbf{cons}(t_1, t_2)$ *is a term.*

We use standard list notation, i.e. we write [ ] for **nil**, and $[t_1, \ldots, t_n]$ with $n > 0$ for $\mathbf{cons}(t_1, [t_2, \ldots, t_n])$. We also use the notation $[\ t_1, \ldots, t_n \mid R\ ]$, defined from $[\ t_1 \mid R\ ] = \mathbf{cons}(t_1, R)$ and $[\ t_1, t_2, \ldots, t_n \mid R\ ] = \mathbf{cons}(\ t_1, [\ t_2, \ldots, t_n \mid R\ ]\ )$.

Data are mapped into lists as follows: Values of primitive types are represented by themselves. Data of compound types without repeated fields are represented by simple lists. Data of compound types with repeated fields are represented by lists of form $[v_1, \ldots, v_n, [\ [w_{1,1}, \ldots, w_{1,m}], \ldots, [w_{k,1}, \ldots, w_{k,m}]\ ]\ ]$, where $v_1, \ldots, v_n$ are the fixed fields, $k \geq 0$ is the size, and $w_{j,1}, \ldots, w_{j,m}$ are the repeated fields. For example a propositional atom of type prop could be represented by $[\ ?\text{var}, [\ ['v'], ['a'], ['r']\ ]\ ]$. Its negation can be represented by $[\ ?\text{neg}, [\ ?\text{var}, [\ ['v'], ['a'], ['r']\ ]\ ]\ ]$. Note that this representation is not the representation that we use at run time. At run time, repeated fields will be stored in C^{++} style vectors. The list representation is intended for reasoning only.

We will now define tree regular expressions. As in the word case, a regular expression z defines a subset of trees, which is called the language of z. We need to define the languages of regular expressions together with the regular expressions. This is not completely trivial, because our regular expressions may contain variables that refer to other regular expressions. These variables are the result of cross references in adjectives, which are allowed. For example in Example 3, the definitions of even and odd refer to each other. As a consequence, we cannot define the meaning of regular expressions at once. Instead we have to define a 'meaning update function', whose fixed point will be the intended meaning of the expression. So, we start by defining regular expressions together with a temporary meaning function σ. The fixed point of the meanings will be defined in Definition 10.

Definition 9. *Assume that* σ *is a partial function that maps variables to tree languages. We recursively define more expressions* z *for which* $\sigma(z)$ *is defined.*

- *If* T *is the name of a primitive type, then* $\sigma(T) = \{x{:}T \mid \top\}$.
- *If* v *is a variable then* $\sigma(v)$ *is defined iff* v *is in the domain of* σ.
- *If* c *is a constant of type* $T =$ bool, char, index, integer, double, *or* selector, *then* $\sigma(c) = \{c\}$ *and* $\sigma(c^{\neq})$ *is defined as* $\{x{:}T \mid x \neq c\}$.
- *If* c *is a constant of type* $T =$ bool, char, index, integer, *or* double, *then we define* $\sigma(c^{>}) = \{x{:}T \mid x > c\}$ *and* $\sigma(c^{<}) = \{x{:}T \mid x < c\}$.
- $\sigma(\mathbf{nil})$ *is defined as* $\{\mathbf{nil}\}$.
- *If* $\sigma(z_1)$ *and* $\sigma(z_2)$ *are defined, then* $\mathbf{cons}(z_1, z_2)$ *is defined as* $\{\mathbf{cons}(t_1, t_2) \mid t_1 \in \sigma(z_1)$ *and* $t_2 \in \sigma(z_2)\}$.
- *If* $\sigma(z_1), \ldots, \sigma(z_n)$ $(n \geq 0)$ *are defined, then* $\sigma(z_1 \wedge \cdots \wedge z_n)$ *is defined as* $\sigma(z_1) \cap \cdots \cap \sigma(z_n)$, *and* $\sigma(z_1 \vee \cdots \vee z_n)$ *is defined as* $\sigma(z_1) \vee \cdots \vee \sigma(z_n)$.

– *If* $\sigma(z_1)$ *and* $\sigma(z_2)$ *are defined, then* $\sigma(\mathbf{cons}^*(z_1, z_2))$ *is defined as the smallest set* S *which has* $\sigma(z_2) \subseteq S$ *and for all* $t_1 \in \sigma(z_1)$ *and* $t_2 \in S$, *we have* $\mathbf{cons}(t_1, t_2) \in S$.

We call the expressions z *for which* $\sigma(z)$ *is defined* regular expressions over σ. *If* $\sigma(z)$ *is defined for some* σ, *we call* z *just a regular expression.*

We define the fixed point operator:

Definition 10. *Let* $\{v_1/z_1, \ldots, v_m/z_m\}$ *be set of simultaneous expression definitions. For every* j $(1 \leq j \leq m)$, *we define* $\sigma_1(v_j) = \emptyset$, $\sigma_{i+1}(v_j) = \sigma_i(z_j)$, $\sigma(v_j) = \bigcup_{i \in \mathcal{N}} \sigma_i(v_j)$.

Example 6. Consider the simultaneous definitions $\{v_1/\mathbf{cons}(1, v_2) \vee \mathbf{nil},\ v_2/\mathrm{cons}(2, v_1)\}$. We have

$$\begin{array}{ll} \sigma_0(v_1) = \emptyset, & \sigma_0(v_2) = \emptyset \\ \sigma_1(v_1) = \{\mathbf{nil}\} & \sigma_1(v_2) = \emptyset \\ \sigma_2(v_1) = \{\mathbf{nil}\} & \sigma_2(v_2) = \{\mathbf{cons}(2, \mathbf{nil})\} \\ \sigma_3(v_1) = \{\mathbf{nil}, \mathbf{cons}(1, \mathbf{cons}(2, \mathbf{nil}))\} & \sigma_3(v_2) = \{\mathbf{cons}(2, \mathbf{nil})\} \end{array}$$

The pattern is clear.

We now give the functions REGEXP that translate type and adjective declarations into regular expressions. For example, Definition 4 can be translated as:

$$\mathrm{nat} = \bigvee \left\{ \begin{array}{l} [\ \mathrm{selector} \wedge (?\mathrm{zero} \vee ?\mathrm{succ}) \wedge ?\mathrm{zero}\ ] \\ [\ \mathrm{selector} \wedge (?\mathrm{succ} \vee ?\mathrm{succ}) \wedge ?\mathrm{succ}, ?\mathrm{nat}\ ] \end{array} \right.$$

Definition 11. *Let* Σ *be a symbol table, and let* T *be a non-compound type.* $\mathrm{REGEXP}_\Sigma(T)$ *returns a regular expression that defines the data terms having type* T.

– *If* T *is a primitive type that is not* void, *then* $\mathrm{REGEXP}_\Sigma(T) = T$.
– *If* T *has form* v, *with* v *an exact identifier, then* $\mathrm{REGEXP}_\Sigma(v) = v$.
– *If* T *has form* v, *where* v *is an inexact identifier for which there exists exactly one type definition* TYPE $v' := T'$ *in* Σ, *with* v' *a possible overload for* v, *then* $\mathrm{REGEXP}_\Sigma(T) = v'$. *Otherwise, the result is an error.*
– *If* T *has form* $T' \circ A$, *then* $\mathrm{REGEXP}(T' \circ A)$ *is obtained as follows: First let* $z_1 = \mathrm{REGEXP}_\Sigma(T')$. *Next, let* $z_2 = \mathrm{REGEXP}_\Sigma(z_1, A)$. *(this refers to Definition 13) Define* $\mathrm{REGEXP}(T' \circ A) = z_1 \wedge z_2$.

Definition 12. *Let* Σ *be a symbol table, let* E *be the expansion set of a compound type that has been processed by the field creation procedure of Definition 7.* $\mathrm{REGEXP}_\Sigma(e)$ *returns a regular expression that defines the expansion* e.

– *Write* e *in the form* $(\overline{v}, \overline{V}, \overline{A}, \overline{w}, \overline{W})$, *let* $n = \|\overline{V}\|$. *We know that the field names in* $\overline{v}$ *and* $\overline{w}$ *are exact.*
– *For every* i $(1 \leq i \leq n$, *construct* $z_i = \mathrm{REGEXP}_\Sigma(V_i \circ A_i)$.
– *If* $\|\overline{w}\| = \|\overline{W}\| = 0$, *then* $\mathrm{REGEXP}_\Sigma(e) = [z_1, z_2, \ldots, z_n]$.

– *If* $\|\overline{w}\| = \|\overline{W}\| > 0$, *then let* $m = \|\overline{W}\|$. *For each* j $(1 \leq j \leq m)$, *let* $y_j = \text{REGEXP}_\Sigma(W_j)$. *(The* W_j *are the types in the repeated part of e.) Define* $\text{REGEXP}_\Sigma(e) = [z_1, z_2, \ldots, z_n,\ \mathbf{cons}^*([y_1, \ldots, y_m], \mathbf{nil})\ ]$.

The translation function for adjectives has three arguments. It needs an additional context Γ, because adjectives may contain field names and references to other adjectives that have to be type checked and resolved. In order to do this, we need to know the type on which the adjective is being applied. This type is represented by Γ. During type checking, we call the decision procedure in Section 3 in order to resolve identifiers.

Definition 13. *Let* Σ *be a symbol table, let* Γ *be regular expression and let* A *be an adjective.* $\text{REGEXP}_\Sigma(\Gamma, A)$ *defines a regular exprssion that represents adjective* A *when applied on type* Γ.

– *If* A *is an inexact identifier* v, *then we need to find an overload for* v. *Let* $v_1, \ldots, v_n$ *be the exact identifiers that have an adjective definition* ADJ $v_i{:}T_i := A_i$ *in* Σ, *s.t.* v_i *is a possible overload of* v, *and* $\sigma(\Gamma) \subseteq \sigma(\text{REGEXP}_\Sigma(T_i))$. *If there exists a unique* i, *s.t. for all* i' $(1 \leq i' \leq n)$ *we have* $\sigma(\text{REGEXP}_\Sigma(T_{i'})) \subseteq \sigma(\text{REGEXP}_\Sigma(T_i))$, *then* $\text{REGEXP}_\Sigma(v) = v_i$. *Otherwise, it is undefined. (We explain the meaning below.)*
– *If* A *has form* $\{c_1, \ldots, c_n\}$*or*$\{c_1, \ldots, c_n\}^C$, *then verify that every* i, *we have* $c_i \in \sigma(\Gamma)$. *Define* $\text{REGEXP}_\Sigma(\Gamma, A) = (c_1 \vee \cdots \vee c_n)$ *and* $\text{REGEXP}_\Sigma(\Gamma, A) = (c_1^{\neq} \wedge \cdots \wedge c_n^{\neq})$.
– *If* A *has form* $(\leq c)$, $(\geq c)$, $(< c)$, *or* $(> c)$, *then first verify that* $c \in \sigma(\Gamma)$. *After that, return the suitable* $c \cup c^{<}$, $c \cup c^{>}$, $c^{<}$, *or* $c^{>}$.
– *If* A *has form* $f[A']$ *and* f *is not exact, then let* $f_1, \ldots, f_n$ *be the exact, non-repeated field names defined in* Σ *that are possible overloads for* f. *Each* f_i *has a definition* FIELD $f{:}U_i(U_{i,1}) := c_i/S_i$ *in* Σ. *If there is an* i *s.t.* $\sigma(\Gamma) \subseteq \sigma(\text{REGEXP}_\Sigma(U_{i,1}))$, *then continue in the next case with* $f_i[A']$. *(there may be none, but not more than one, because field names cannot be redefined on a single expansion) then*
– *If* A *has form* $f[A']$ *and* f *is exact, then let* FIELD $f{:}U(U_1) := c/S$ *be the definition of* f. *Let* $z = \text{REGEXP}_\Sigma(U \circ A')$. *(We recursively obtain an expression for* A' *on type* U.*) Define* $\text{REGEXP}_\Sigma(\Gamma, A) = \mathbf{cons}^{c-1}(\top, \mathbf{cons}(z, \top))$. *It is a list of length at least* c, *whose* c*-th element equals* z.
– *If* A *has form* $\{c_1, \ldots, c_n\}[A']$, $\forall^{\geq c}[A']$, $\text{size}(\{c_1, \ldots, c_n\})$, *or* $\text{size}^{\geq}(c)$, *then we first resolve the* repbase *field in context* Γ. *Let* $f_1, \ldots, f_n$ *be the exact field names defined in* Σ, *that are candidate overloads for* repbase, *and whose definition* FIELD$^{[]}$ $f{:}U_i(U_{i,1}, \text{index}) := c_i/1$ *has* $\sigma(\Gamma) \subseteq \sigma(\text{REGEXP}_\Sigma(U_{i,1}))$. *If* $n = 1$, *then* f_1 *is the overload for* repbase.
  1. *If* $A = \{c_1, \ldots, c_n\}[A']$, *then let* $z = \text{REGEXP}^{[]}_\Sigma(\Gamma, A')$, *where* $\text{REGEXP}^{[]}$ *is defined in Definition 14. Let* $m = \max(\{c_1, \ldots, c_n\})$. *Define* $\text{REGEXP}_\Sigma(\Gamma, A) = \mathbf{cons}^*(\top, [z_1, \ldots, z_m])$, *where* $z_j = z$ *if* $j \in \{c_1, \ldots, c_n\}$, *and* $\top$ *otherwise.*
  2. *If* $A = \forall^{\geq c}[A']$, *then let* $z = \text{REGEXP}^{[]}_\Sigma(\Gamma, A')$. *Define* $\text{REGEXP}_\Sigma(A) = \mathbf{cons}^*(z, z')$, *where* $z' = [\top, \ldots, \top]$ *with length* c.

3. *If* $A = \text{size}(\{c_1, \ldots, c_n\})$, *then for each* i, *let* $z_i = [\top, \ldots, \top]$ *with length* c_i. *Define* $\text{REGEXP}_\Sigma(A) = z_1 \vee \cdots \vee z_n$.
4. *If* $A = \text{size}^{\geq}(c)$, *then let* $z = [\top, \ldots, \top]$ *with a length of* c. *Define* $\text{REGEXP}_\Sigma(A) = \text{cons}^*(\top, z)$.

– *If* A *has form* $A_1 \cup \cdots \cup A_n$, *then* $\text{REGEXP}_\Sigma(A) = \text{REGEXP}_\Sigma(A_1) \vee \cdots \vee \text{REGEXP}_\Sigma(A_n)$.
– *If* A *has form* $A_1 \cap \cdots \cap A_n$, *then iteratively let* $z_i = \text{REGEXP}_\Sigma(\Gamma \wedge z_1 \wedge \cdots \wedge z_{i-1}, A_i)$. *Return* $z_1 \wedge \cdots \wedge z_n$.

Definition 14. *Let* Σ *be a symbol table, let* Γ *be a regular expression and let* A *be an adjective.* $\text{REGEXP}^{[]}_\Sigma(\Gamma, A)$ *defines a regular expression that represents the (repeated) adjective* A *when applied on type* Γ.

– *If* A *has form* $f[A']$, *with* f *a non-exact field name, and* A' *a (non-repeated) adjective, then we need to find an overload for* f. *Let* $f_1, \ldots, f_n$ *be the exact (repeated) field names in* Σ *that are potential overloads of* f. *Each* f_i *has a definition* $\text{FIELD}^{[]}$ $f_i{:}U_i(U_{i,1}, \text{index}) := (c_i, d_i)$ *in* Σ. *If there is an* i, *s.t.* $\sigma(\Gamma) \subseteq \sigma(\text{REGEXP}_\Sigma)(U_{i,1}))$, *then continue in the next case with* $f_i[A']$. *As in the non-repeated case,* $n \geq 1$.
– *If* A *has form* $f[A']$, *with* f *an exact name of a repeated field, then let* $\text{FIELD}^{[]}$ $f{:}U(U_1, \text{index}) := (c, d)$ *be the definition of* f. *Let* $z = \text{REGEXP}_\Sigma(U \circ A')$. *Define* $\text{REGEXP}^{[]}(A) = \mathbf{cons}^{d-1}(\top, \mathbf{cons}(z, \top))$. *It is a list of length at least* d, *whose* d*-th element equals* z.

We show how natural numbers are translated into Horn clauses. The expansion set of nat was given in Example 4.

Example 7. We define the regular expressions for propositional logic:

$$\text{prop} = \bigvee \begin{cases} [\text{ selector}\wedge\text{?var, } \mathbf{cons}^*(\text{ [char]}, \mathbf{nil}\text{)]} \\ [\text{ selector}\wedge\text{?not, prop]} \\ [\text{ selector} \wedge (\text{?implies})\vee\text{?equiv), prop, prop]} \\ [\text{ selector} \wedge (\text{?and}\vee\text{?or}), \mathbf{cons}^*(\text{ [prop]}, \mathbf{nil}\text{)]} \end{cases}$$

Negation normal form (NNF) can be defined by the following adjective:

$$\text{nnf} = \bigvee \begin{cases} [\text{ selector}\wedge\text{?var} \mid \top\text{]} \\ [\text{ selector}\wedge\text{?not, [selector}\wedge\text{?var} \mid \top\text{]} \mid \top\text{]} \\ [\text{ selector} \wedge (\text{?and}\vee\text{?or}), \mathbf{cons}^*(\text{ [nnf]}, \mathbf{nil}\text{) } \mid \top\text{]} \end{cases}$$

3 Decision Procedure for Type Inclusion

Our goal is to decide adjective inclusion relations of form (nat $\subseteq$ even $\vee$ odd), (case completenss) or (nat $\wedge$ even) $\wedge$ odd $\subseteq \bot$ (case disjointness.) More precisely

Definition 15. *Let* $\{v_1/z_1, \ldots, v_n/z_n\}$ *be a set of regular expression definitions. Let* z'_1 *and* z'_2 *be regular expressions over the variables* $v_1, \ldots, v_n$. *Does* $\sigma(z'_1) \subseteq \sigma(z'_2)$ *hold? If not, find a term* $t \in \sigma(z'_1) \backslash \sigma(z'_2)$.

The type inclusion problem can be solved by adapting standard tree automata techniques. We will translate the definitions of the regular expressions z_1', z_2' into non-deterministic tree automata. After that, we transform the resulting automata into deterministic automata, and solve the inclusion problem by constructing the product automaton.

Problem 15 is also used in Definitions 13 and 14 for the resolution of inexact identifiers in adjective definitions. We define non-deterministic tree automata, mostly following [2].

Definition 16. *A* non-deterministic, merging, finite tree automaton *(NFTA) is defined as a tuple* $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$, *where* Q *is a finite set of states,* $\mathcal{F}$ *is the signature,* $Q_f \subseteq Q$ *is the set of accepting states, and* Δ *is a set of transitions of two possible forms: The first form is* $f(q_1, \ldots, q_n) \to q$, *with* $f \in \mathcal{F}$, $q_1, \ldots, q_n, q \in Q$, *and the second form is* $q_1, \ldots, q_n \to q$ *with* $n \geq 1$.

Transitions of the first form are completely standard (see [2]). Transitions of the second form are ϵ-transitions when $n = 1$. If $n > 1$, they are truly merging, because they require the automaton to be in all states $q_1, \ldots, q_n$ for the same term, in order to reach state q. Merging transitions naturally occur as the translations of regular expressions of form $z_1 \wedge \cdots \wedge z_n$. In our case, $\mathcal{F}$ consists of the symbols **cons**, **nil** as well as symbols of form $c, c^{\neq}, c^{<}, c^{>}$. Symbols of the last three forms need special attention, because they have hidden dependencies originating from their meaning. Without special treatment, e.g. $3^{>} \subseteq 2^{>}$ would be not provable. We will deal with this problem in Definition 19.

Definition 17. *Let* $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ *be a tree automaton. A* configuration *of* $\mathcal{A}$ *has form* $q(t)$ *with* t *a term over signature* $\mathcal{F}$ *and* $q \in Q$. *The set of* reachable configurations $\mathcal{R}$ *is defined as the smallest for which*

- *If* $q_1(t_1), \ldots, q_n(t_n) \in \mathcal{R}$, *and* $(\ f(q_1, \ldots, q_n) \to q\ ) \in \Delta$, *then* $q(f(t_1, \ldots, t_n)) \in \mathcal{R}$,
- *If* $q_1(t), \ldots, q_n(t) \in \mathcal{R}$, *and* $(\ q_1, \ldots, q_n \to q\ ) \in \Delta$, *then* $q(t) \in \mathcal{R}$.

An automaton A accepts t *if there exists a state* $q \in Q_f$, *s.t.* $q(t) \in \mathcal{R}$.

Regular expressions can be translated into tree automata:

Theorem 1. *Let* $\{v_1/z_1, \ldots, v_n/z_n\}$ *be a set of regular expression definitions, possibly referring to each other.*

It is possible to construct non-deterministic, finite tree automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$, *s.t. for every tree* t, $t \in \sigma(z_i) \Leftrightarrow \mathcal{A}_i$ *accepts* t.

Theorem 1 is a minor variation of a standard result. The set of definitions $\{v_1/z_1, \ldots, v_n/z_n\}$ can be viewed as a set of regular equations (Section 2.3 in [2]). In order to translate $\wedge$, merging transitions can be used.

Definition 18. *Let* $\mathcal{A}$ *be an NFTA. We call* $\mathcal{A}$ *a* deterministic finite tree automaton *(DFTA) if* $(\ f(q_1, \ldots, q_n) \to q\ )$, $(\ f(q_1, \ldots, q_n) \to q'\ ) \in \Delta$ *implies* $q = q'$, *and* Δ *contains no transitions of the second type.*

Theorem 2. *Every NDFA $\mathcal{A}$ can be transformed into a DFTA.*

The proof is completely standard. It is analogous to the subset construction for word languages [10, 1]. The procedure is not affected by the presence of merging transitions.

The domain of discourse defines the set of predicates that could play a role in proving F. Note that it is not possible that there are conflicts in $\mathcal{F}$ which would result in the proof of F using predicates that are not in Π.

Definition 19. *Let $\mathcal{A}_1, \mathcal{A}_2$ be two DFTAs obtained by translation of regular expressions. We want to determine whether $\mathcal{A}_1$ accepts t implies $\mathcal{A}_2$ accepts t. We proceed as follows:*

- *For every type $T \in \{$ bool, char, index, integer, double $\}$, do the following: Let $\{c_1, \ldots, c_n\}$ be the set of values of type T for which $\Delta_{1,2}$ contains a transition of form $c_i(q) \to q'$, $c_i^{\neq}(q) \to q'$, $c_i^{>}(q) \to q'$, or $c_i^{<}(q) \to q'$. Assume that $c_1, \ldots, c_n$ are sorted in increasing order. We extend the signatures $\mathcal{F}_{1,2}$ by the following symbols: $c_1^{<}$ with meaning 'less than c_1', $\langle c_i, c_{i+1}\rangle$ with meaning 'strictly between c_i and c_{i+1}', and $c_n^{>}$ with meaning 'greater than c_n'.)*
  Next, replace all transitions of form $c^{\neq}(q) \to q'$ by $c^{<}(q) \to q'$ and $c^{>}(q) \to q'$. After that, for all $i > 1$, replace the transitions of form $c_i^{<}(q) \to q'$ by

$$\{c_{i'}(q) \to q' \mid i' < i\} \cup \{\langle c_{i'}, c_{i'+1}\rangle(q) \to q' \mid 1 < i' < i\} \cup \{c_1^{<}(q) \to q'\}.$$

  Similarly, for all $i < n$, replace the transitions of form $c_i^{>}(q) \to q'$ by

$$\{c_{i'}(q) \to q' \mid i' > i\} \cup \{\langle c_{i'}, c_{i'+1}\rangle(q) \to q' \mid i \leq i' < n\} \cup \{c_n^{>}(q) \to q'\}.$$

  Note that some of the intervals can be empty, for example $\langle 3, 4\rangle$ or $\mathbf{f}^{<}$. In that case, the transition should not be added. Emptyness can be checked easily.
- *For the type* selector, *let S be the set of selector values that occur in $\Delta_{1,2}$. Replace every transition of form $s^{\neq}(q) \to q'$ by*

$$\{s'(q) \to q' \mid s' \in S \wedge s' \neq s\}.$$

After that, we can use a standard approach: Construct the product automaton $\mathcal{A}_1 \times \mathcal{A}_2$, and check that for every state $(q_1, q_2) \in Q_1 \times Q_2$ $q_1 \in Q_{f,1} \Rightarrow q_2 \in Q_{f,2}$.

4 Conclusions and Future Work

We provided a type system that allows concise definitions of recursive structures frequently occurring in logic. The system allows definition of subtypes which we call 'adjectives'. We prefer the name 'adjective' over 'subtype' because most adjectives have non-permanent nature. They are used once in a case analysis, while another case analysis may use a different partition.

We are not sure about the borders of our approach. At this moment, adjectives are strictly unary, i.e. properties of a single term. It is likely that by

using suitable product constructions, some useful binary relations (for example equality) can also be expressed by means of tree automata. We want to find the border between what is useful, and what can be decided by means of tree automata.

We did not discuss how to decide type membership for individual terms. This is needed for type checking concrete terms. This can be done in principle by straightforwardly applying Definition 17, but there may be better approaches that take context into account. We are not sure what is the best approach for type checking terms, and this future research.

References

1. Alfred Aho and Jeffrey Ullman. *Principles of Compiler Design*. Addison-Wesley Publishing Company, 1977.
2. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2007. release October, 12th 2007.
3. P. Cousot and R. Cousot. Abstract interpretation: A unified lattic model for static analysis of programs by construction or approximation of fixpoints. In Geoff Sutcliffe and Andrei Voronkov, editors, *POPL*, pages 238–252. ACM Press, 1977.
4. Hans de Nivelle. theorem prover Geo III. Can be obtained from `http://www.ii.uni.wroc.pl/~nivelle/software/` or from `http://www.tptp.org/CASC/`, 2015-2019.
5. Hans de Nivelle and Jia Meng. Geometric resolution: A proof procedure based on finite model search. In John Harrison, Ulrich Furbach, and Natarajan Shankar, editors, *International Joint Conference on Automated Reasoning 2006*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 303–317, Seattle, USA, August 2006. Springer.
6. Graham Hutton. *Programming in Haskell (second edition)*. Cambridge University Press, 2016.
7. Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml (functional programming for the masses)*. O'Reilly, 2013.
8. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 2005.
9. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala Third Edition*. Artima Press, 2007-2016.
10. Michael Sipser. *Introduction to the Theory of Computation (Third Edition)*. CENGAGE Learning, 2013.