Subtyping Arithmetical Types

Joseph (Yossi) Gil*†

Systems and Software Development Laboratory
Department of Computer Science
Technion—Israel Institute of Technology
Haifa 32000, Israel

Abstract

We consider the type system formed by a finite set of primitive types such as integer, character, real, etc., and three type construction operators: (i) Cartesian product, (ii) disjoint sum, and (iii) recursive type definitions. Type equivalence is defined to obey the arithmetical rules: commutativity and associativity of product and sum and distributivity of product over sum. We offer a compact representation of the types in this system as multivariate algebraic functions. This type system admits two natural notions of subtyping: "multiplicative", which roughly corresponds to the notion of object-oriented subtyping, and "additive", which seems to be more appropriate in our context. Both kinds of subtyping can be efficiently computed if no recursive definitions are allowed. Our main result is that additive subtyping is undecidable in the general case. Perhaps surprisingly, this undecidability result is by reduction from Hilbert's Tenth Problem (H10): the solution of Diophantine equations.

1 Introduction

Central to the object-oriented (OO) paradigm is the notion of *subtyping*. Central to many modern programming languages is the notion of *recursive data types*. The combination of these two notions is a fascinating and difficult topic. Indeed, the question of subtyping recursive types was open for many years until settled by the seminal work of Amadio and Cardelli [1] which gave the first algorithm for subtyping recursive types. Consequently, the run time of this algorithm was improved from exponential to polynomial by Kozen, Palsberg and Schwartzbach [26]. Subtyping of recursive types was used in type inference systems such as [40] and [14]. A theoretical foundation was laid out by Brandt and Henglein [9] who presented a sound and complete axiomatization of the coinductive inference that these algorithms use.

In this paper, we revisit this question, asking how could it be married with the notion of *structural equivalence*, as opposed to *name equivalence*, which is crucial to distributed computation, and more generally, *structural subtyping*. We use structural equivalence in its deepest sense, which includes application of commutative, associative and distributive rules on types.

Thus, the problem that this paper deals with can be labeled as *distributive and commutative structural subtyping of recursive types*. The research presented here is of a theoretical, foundational sort, aimed at a better understanding of the notion of subtyping and the possible variations of this concept. Nevertheless, the interest in this problem is not merely a matter of academic curiosity. In fact, the problem was raised and brought to our attention in the course of implementing an IBM experimental product, nicknamed "Mockingbird" [5, 4].

Mockingbird is a prototype tool for developing inter-language and distributed applications. It facilitates data exchange and communication between applications which may be written in different programming languages. The Mockingbird type compiler reads two type definitions, which may be user annotated, and tries to generate a stub to convert instances of one type into instances of the other. Thus, given two types, T_1 and T_2, the type compiler is concerned with the following classical questions: "are T_1 and T_2 equivalent?", "is T_1 a subtype of T_2" (if the answer to the previous question is negative), and "how should a conversion between T_1 and T_2 be carried out?" (if the answer to any one of the previous questions is positive).

These three questions are parameterized by the type system from which T_1 and T_2 are drawn and by the definition of equivalence and subtyping. Intuitively, saying that T_1 is a subtype of T_2 means that values of T_1 can be used wherever values of T_2 can be used. This leads to several distinct meanings to the "subtype" term.

Set inclusion If we consider types T_1 and T_2 as the set of all their values, saying that T_1 is a subtype of T_2 could mean that every value of T_1 *is also a value of* T_2. This is the notion of subtyping in languages such as Ada [22].

Value coercion Yet another notion of subtyping is that every value of T_1 can be *coerced into a value of* T_2. This is the typical case in OO databases in which a *relation* is considered a subtype of another relation which has fewer columns. This is also the definition of subtyping in many OO languages. The type Manager may have more attributes than its supertype Employee. By removing or ignoring these extra attributes, every "manager" can be considered to be an "employee".

Interface extension In many OO languages the subtype notion is further extended to demand that the set of routines (functions and procedures) defined in a subtype, i.e., the interface of the subtype, is a superset of the interface of the supertype.

*Work done in part while the author was at the IBM T. J. Watson Research Center.
†yogi@CS.Technion.CS.IL

POPL '01 1/01 London, UK
 ISBN 1-58113-336-7/01/0001...$5.00

In this paper we concentrate mostly on the first two notions of subtyping. In the context of data interchange between applications, it is more important to capture the structure of the information, rather than the behavior associated with it internally in each application. Indeed, languages designed for that purpose, such as ASN.1 [44], EDIFACT and Sun's XDR language for remote procedure call, do not associate behavior with data types, just like many non-OO languages. To use the C++ [41] terminology, data types in our system do not have "function members". It should be clear that this restriction does not invalidate the notion of subtyping. Indeed, ASN.1 has a notion of subtyping, albeit not as general as it could be.

Yet another motivation for structural equivalence comes from the use of types as search keys in software libraries [35]. The type system used in that work is geared toward functional programming. As a result, $Th^1_{\times\mathbf{T}}$, the type system used there, includes the function type operator, which renders it very different from ours.

An important class of examples motivating our type systems comes from object oriented compiler compilers, such as SOOP [17], YOOC and TROOPER [6, 7], which generate an OO type definition from a given BNF grammar, and a parser that converts a word in the language defined by the BNF into an instantiation of this type definition. The correspondence between a BNF and a data type definition is based on the following principles. Grammar terminals such as keywords, which can only occur in one form in the input, are ignored, while terminals such as numerical constants and identifiers are represented as primitive data types. Nonterminals correspond to composite data types, where concatenation in the body of a BNF production is interpreted as a record definition, while alternative productions are realized by a choice type operator. Again, these data description languages lack behavior.

We argue that a deeper understanding of the structural, nonbehavioral, subtyping is important also for the seemingly more general notion subtyping as used in contemporary OO languages, which includes both the structural and the behavioral aspects of subtyping. First, an intractability result of structural subtyping, of the sort we provide here, also implies the intractability of any notion of subtyping which includes this kind of structural subtyping. Conversely, algorithms for a particular kind of structural subtyping can be used as subroutines in algorithms for more general subtyping problems.

Secondly, in this paper we develop a unique mathematical theory that gives a representation in the form of generating functions and power series to a rather general family of types. The attempt to represent types as power series leads to a discovery of at least one family of degenerate types that might indicate programming errors, and can be detected automatically.

We call attention to the fact that *interface extension* in one of its forms is reduced again to structural subtyping. Specifically, it is often the case that the correspondence between the routines in the supertype and those in the subtype is not required to be exact, and that every routine in the supertype must have a *compatible* routine in the subtype. Compatibility traditionally means that the type of a routine in the subtype is a subtype of the type of the corresponding routine in the supertype. Since the subtyping of routines is in its turn defined in terms of subtyping relations between the type of their arguments (and the return type in case of functions), difficult issues of recursive subtyping are raised. This sort of mutual recursion was the main challenge that Amadio and Cardelli met in their subtyping algorithm, and will not concern us as much here. (As we will see, even without the subtyping of routines, the subtyping problem remains interesting, and as we show, becomes even more difficult, by the introduction of what may be called *arithmetical rules*.) We may think of a different slant on interface extension, which might be called *the structural interface extension problem*, in which it is required to match the signatures of routines in the subtype with the signatures of routines in the supertype, while allowing arbitrary renaming of routines, their arguments as well as reordering of arguments.

The thrust of this paper is the definition of two type systems and type equivalence and subtyping relationships in these systems, and a study of the arithmetical rules in these systems. The type system $\mathcal{P}$ is formed by a finite set of primitive types such as integer, character, real, etc., and two type construction operators: Record and Choice. Types in $\mathcal{P}$ will be encoded as multivariate polynomials. Hence, $\mathcal{P}$ is also called the set of polynomial types.

The type system $\mathcal{A}$ is the same as $\mathcal{P}$ except that it includes an operator for recursive type definitions. Types in $\mathcal{A}$ are especially suited for modeling data type definitions (DTD) in SGML [18], and its rapidly growing offspring XML [19], ASN.1 recursive types, as well as OO compiler compilers. Types in $\mathcal{A}$ will be encoded as formal power series, which can also be thought of as multivariate algebraic functions.

We consider two natural definitions of subtyping in these systems: *Multiplicative subtyping*, which roughly corresponds to value coercion kind of subtyping, can be solved using polynomial division in $\mathcal{P}$ and using polynomial elimination and GCD in $\mathcal{A}$.

The second variant, called *additive subtyping*, corresponds to set inclusion. Although it is efficiently computable in $\mathcal{P}$, it turns out to be undecidable in $\mathcal{A}$. In fact, it is undecidable even in a type system $\mathcal{R}$, nicknamed the *rational types*, where $\mathcal{P} \subset \mathcal{R} \subset \mathcal{A}$. Perhaps surprisingly, this undecidability result is by reduction from Hilbert's Tenth Problem (H10): the solution of Diophantine equations.

This work deals with the decision problem. The third question asked by the Mockingbird type compiler, namely the construction and the study of a type converter, is left for future research, although we do provide a sketch here of how this might be done in an orderly fashion.

Section 2 informally describes our type systems. The description is continued in Section 3 which describes how values in this type system are structured, and provides intuition behind our definitions of type equivalence and subtyping. Section 4 presents the type system $\mathcal{P}$, and discusses the notions of type equivalence and the two main notions of subtyping in it. Algebraic types and their representation as formal power series are discussed in Section 5. Formal definitions of $\mathcal{A}$ and $\mathcal{R}$ are the subject of Section 6. Sections 7 and 8 give the proof that subtyping in $\mathcal{R}$ is undecidable. Section 7 is the more technical one, and may be skipped in first reading. In contrast, the specimens of arithmetical types presented in Section 8 may be of independent interest. Section 9 concludes with some open problems.

2 Overview of the type system

This section provides an informal overview of our type systems, concentrating in polynomial types, and in explaining the arithmetical rules.

Compare for example the Pascal [43] type definition of Figure 1 with that of C [25] in Figure 2. To a human, it is obvious that both definitions denote a binary tree with an integer data field in each node. To reach the same conclusion, the Mockingbird type compiler must ignore syntactical differences between languages (standard practice in the study of type systems), the names of fields (as usual in structural equivalence), the order of definitions in a record, and assume that the corresponding primitive types in the two languages are interchangeable (the working hypothesis of ASN.1 and other data interchange languages). It is tacitly assumed that there is no subtyping relationship between two distinct primitive types.

```
TYPE
T1 = ^Y;
Y = Record
    left, right:  T1;
    value:  Integer
end
```

Figure 1: A binary tree node in Pascal.

```
typedef struct X {
    int data;
    struct X *rchild;
    struct X *lchild;
} *T2;
```

Figure 2: A binary tree node in C.

To facilitate comparison of types between different languages, we assume that the type system has a set of primitive types, and that no two distinct primitive types are interchangeable. In comparing types we view all occurrences of a primitive type as being equivalent. When this is not the desired behavior, new primitive types can be introduced, in a manner similar to *branding* in Modula-3. Mockingbird effectively implements branding by means of user annotations, which may mark some of C's **typedef**'s as being new primitive types.

To model C's **struct**, Pascal's **Record**, and other similar constructs, the type system will have a *product type construction operator*. (We defer the introduction of notation for the type construction operators until Section 4.)

As the example in figures 1 and 2 shows, there is a good reason to make product a *commutative* operation. This demand does not show in [1]. *Associativity* of product is also required to facilitate deep comparisons.

Our type system admits also the *sum type construction operator*, also known as *choice*, for representing e.g., **union** in C, *variant records* in Pascal, or ML [34] constructors. Consider for example Figure 3, which represents the requirement of the BIBTEX [33, 28] bibliographic system that a book has either an author or an editor, but never both.

We tacitly assume that there is some kind of a mechanism of designating which option is selected in a choice. Since a pointer or reference to a data type X is either nil *or* it has a value of type X, we represent such a pointer as a choice between X and **Unit**, the empty product. It is always possible to determine for a pointer which of the options of it was selected. Just like in ASN.1, SGML, SOOP, etc., but also pure Lisp, we cannot represent data structures in which more than one pointer may point to the same data item.

Note that with the commutative and associative rules the product operator is nothing more than a multi-set of the types on which it operates.

```
struct {
    union {
        struct Editor editor;
        struct Author author;
    } c;
    struct Volume_Data data;
} Book;
```

Figure 3: An example of using sum (choice) type operator in C.

```
union {
    struct {
        struct Author author;
        struct Volume_Data data;
    } with_author;
    struct {
        struct Editor editor;
        struct Volume_Data data;
    } with_editor;
} Book_Alternative;
```

Figure 4: An alternative definition of Figure 3.

An enumerated type is a choice between several of **Units**. Since field names are insignificant, the names of enumerated values are insignificant as well: Two enumerated types of the same length are considered equivalent. Branding can be used if this is not the desired effect.

For the sake of completeness, we also introduce the type **None**, which can be thought of as an empty choice. This type, sometimes called *bottom type* and denoted as $\perp$, has no legal values at all. It serves as the neutral element of sum. Also, including **None** in any product will render the result **None** as well.

Java [2], Eiffel [31] and many other OO languages do not include a *general purpose* choice; choice there is restricted to pointers and to enumerated types. It is further research to extend the undecidability result to such systems (see also Section 9 below).

We assume that sum is *commutative* and *associative*. Thus, a choice between A and B is the same as a choice between B and A. Also, a choice between two given enumerated types is the same as an enumerated type whose number of values is the same as the sum of values in the given types.

The *distributive rule of product with sum* also applies. Thus, the differences between the types defined in Figure 3 and in Figure 4 are considered a matter of personal preferences that should be overlooked in type comparisons.

Let us use the term *arithmetical rules* as a collective term for associativity and commutativity of sum and product and distributivity of product with sum. Arithmetical rules also include the existence of **Unit** and **None** as neutral elements for product and sum, alternatively defined as the corresponding compound types with zero arguments. It is also convenient to use the special behavior of **None** in product.

Types formed by the primitive types by means of sum and product, and where the arithmetical rules apply are called the *polynomial types*. The *algebraic types* are the same as the polynomial types except that recursive definitions are allowed. A generic name for both type systems is *arithmetical types*.

We should note that Mockingbird aspirations transcend arithmetical types. Beyond these, [3] discusses function types, *ports* (which are a mechanisms to express call-by-reference) and *dynamic types*, (which are used to express OO polymorphism). It should also be mentioned that the polynomial types are a subset of what may be called *Tarski's types*, which extend the algebraic type-system with the function type operator, obeying rules similar to that of exponentiation (see [16, Chap. 1.8.3]).

3 Values of Arithmetical Types

Since the arithmetical types are composed of primitive types, any instance (or value) of a compound arithmetical type comprises

value assignments to its primitive types components. In case composition was by a product type operator, the instance is simply a tuple of assignments to the operands. Exactly one choice must be taken in each sum operator. Therefore, the value assignments to any primitive types can be represented simply as a multi-set. A multi-set of this sort could include for example two integer values, three floating values and a value of an enumerated type which was designated as a primitive type by branding. We call this multi-set, when the values are omitted and just the types remain, a *configuration*. A configuration is nothing but a product of primitive types, which, as we shall see below, will be encoded as a *monomial*.

It is important to stress at this stage that the level of abstraction imposed by the arithmetical rules *dictates* this unordered multi-set perspective. Consider for example a type U_1 which is a product of two integers. Then, a value of U_1 is an *unordered* set of two integer values. The match between these two values and the fields of U_1 is not part of our type system. As mentioned above, branding is an easy means to distinguish between occurrences of a primitive types. The type U_1 however, is the data type of a multi-set of two integers.

If the definition of an arithmetical type uses a choice, then the same configuration can be obtained in several ways. A value therefore must contain not only the values of a certain configuration, but also a designation of a configuration among others. In presence of the arithmetical rules, and most specifically, commutativity, the matter of designating one configuration among others takes some pondering.

To highlight the issue of commutativity of summation, consider a type U_2 defined as a choice between two integers. Type U_2 gives rise to two identical configurations, each with a single integer. A value of U_2 is therefore an integer, marked with a tag drawn from the set (say) $\{1, 2\}$. This tagging is *not* used for matching this integer against a specific field in U_2, since roughly speaking U_2 has only one field, which may take two different "kinds" of integers. Tagging is used however to make a distinction between values; integers 53_1 and 53_2 are two distinct values of the single "field" of U_2.

In the same fashion, a value of the binary tree data type in Figure 1 could be a multi-set of three integer values, with marked with a tag drawn from a set of five elements which correspond to the five different binary trees with three nodes.

Still, the mapping between these three integers and the four nodes in a tree is not part of our type system. As can be seen from the binary tree example, the number of different configurations can in general be infinite when recursive type definitions are allowed. We will consider two types as being *equivalent* if they generate the same number of configurations of each kind. This definition is almost a direct consequence of the structural equivalence demands that we have made. Similarly, we consider a type T a *subtype* of another type T', if the (potentially infinite) multiset of configurations that T generates is contained in the multiset of configurations that T' generates.

Let us describe now an abstract device, *a canonical configuration generator* or CCG, which given an arithmetical type, will produce all configurations of this type, together with their count. A CCG receives as input an abstract syntax tree of a type definition *prior* to the application of the arithmetical rules, or a representation as a polynomial. This abstract syntax tree is ordered, which makes it possible for the CCG to produce all occurrences of each possible configuration of a type by applying a breadth-first left-to-right scan of this tree. Thus, the CCG generates a stream of configurations, and the configuration matching can be done by mapping the first occurrence of a certain configuration in T against the first occurrence of the same configuration in T'.

Furthermore, CCG is readily generalized to deal with a *regular* infinite abstract tree, i.e., the abstract syntax tree of a recursive arithmetical type. It is plausible that the canonical configuration ordering produced by a CCG correlates with human intuition, since the working of a CCG can be thought of as retrace the process by which a human understands complicated type definitions.

The CCG imposes a deterministic matching between the configurations of one type and the configurations of another type. Thus, when one does a there is a unique, deterministic matching between the configurations of one type and the There are still a number of open questions regarding CCGs. For example, it is useful to have an algorithm which would extract the i^{th} occurrence of a certain configuration from a regular infinite abstract syntax tree in an efficient manner, i.e., without applying the full scan.

At this point, a weary programmer might complain that is would not be intuitive to program in a system that provides all these arithmetical conversions. The answer is that as is, the type system is not usually used internally in a program, but rather for the purpose of producing an automatic conversion between foreign data representations relying on user annotations. It should also be commented here that the FiSh programming language [23] designed for combining high efficiency with high level abstraction makes a similar distinction between "shape" and "content".

A requirement that came from Mockingbird users was to maintain some kind of "structured mapping" between configurations of one type with configurations of another type. The problem of satisfactorily defining what such a mapping may mean, while still maintaining the arithmetical rules, appears to be illusive and difficult, and is left for further research.

A natural question which arises here is what happens if we forgo the distinction between configurations of the same structure. In other words, a value of a certain type will cease to carry the configuration tag. Such a structure would add an idempotent summation rule to our arithmetical rules. As it turns out, the problems of subtyping and of type equivalence can be done in polynomial time [27, Chap. 9]. Similarly, one may also wonder about type equivalence and subtype in a type system that does not admit commutativity of product. In this case, although type equivalence is still computable, subtyping is undecidable even if there are only two primitive types [38].

4 Polynomial Types

4.1 Notations and Conventions

We will refer to non-negative integers as *natural numbers*. Let $\mathbb{N}$ be the commutative semi-ring[1] of natural numbers. The commutative semi-ring $\mathbb{B}$ of Boolean values, where 0=False, 1=True, and OR as addition and AND as multiplication will also be of interest.

As before, types will usually be denoted using upper case letters. An extensive use of formal variables will be made. These will usually be denoted by lower case letters. Sets of variables are denoted by boldface letters: $\mathbf{x}$, $\mathbf{y}$, etc. Let $\mathbf{x} = \{x_1, \ldots, x_k\}$ be a set of variables. Then, $[\mathbf{x}]$ denotes the set of *monomials* of $\mathbf{x}$, i.e., all expressions of the form

$$\prod_{i=1}^{k} x_i^{n_i} \tag{1}$$

where $n_i \in \mathbb{N}$. Note that monomials may not have coefficients. Still, $1 = \prod_{i=1}^{k} x_i^0 \in [\mathbf{x}]$.

[1] A *commutative semi-ring* is a set with constants 0 and 1, addition and multiplication operations which obey the usual commutative, associative and distributive rules. A semi-ring does not necessarily have subtraction or division.

For encoding a type system in which the commutative rule of multiplication does not apply, it is convenient to use *pseudo-monomials* which are nothing but the set of all finite strings of formal variables. The set of all pseudo-monomials over $\mathbf{x}$ is denoted by $\mathbf{x}^*$. Multiplication in $\mathbf{x}^*$ is simply string concatenation, which is non-commutative.

Let $\mathbb{A}[\mathbf{x}]$ denote the set of all formal polynomials over $\mathbf{x}$ with coefficients in $\mathbb{A}$, i.e., finite linear combinations of the elements of $[\mathbf{x}]$ with coefficients in $\mathbb{A}$. For a polynomial $p \in \mathbb{A}[\mathbf{x}]$ and a monomial $m \in [\mathbf{x}]$, let $[m]p \in \mathbb{A}$ denote the coefficient of m in p. Also, let $\mathbb{A}[[\mathbf{x}]]$ denote the set of *formal pseudo-polynomials* over $\mathbf{x}$, which is finite linear combinations of $\mathbf{x}^*$. In carrying out computations in $\mathbb{A}[[\mathbf{x}]]$ we allow members of $\mathbb{A}$ to commute with members of $\mathbf{x}^*$.

4.2 Polynomial Encoding

Let $Z = \{z_1, \ldots, z_n\}$ be a set of elements, which will be thought of as our primitive types. A type system which includes product and choice over Z and the special types **Unit** and **None** is defined by the following BNF:

$$\begin{aligned} \tau &::= z_1 \mid \cdots \mid z_n \\ \tau &::= \mathbf{Unit} \mid \mathbf{None} \\ \tau &::= \mathit{choice}(\tau, \tau) \mid \mathit{product}(\tau, \tau) \end{aligned} \tag{2}$$

Type equality is defined as the minimal symmetrical and transitive relation obeying the following axioms

$$\begin{aligned} &\mathbf{None} \approx \mathbf{None}; \quad \mathbf{Unit} \approx \mathbf{Unit} \\ &\forall z \in Z \bullet z \approx z \\ &\mathit{choice}(\mathbf{None}, \tau) \approx \tau; \quad \mathit{product}(\mathbf{Unit}, \tau) \approx \tau \\ &\mathit{product}(\mathbf{None}, \tau) \approx \mathbf{None} \\ &\mathit{choice}(\tau_1, \tau_2) \approx \mathit{choice}(\tau_2, \tau_1) \\ &\mathit{product}(\tau_1, \tau_2) \approx \mathit{product}(\tau_2, \tau_1) \\ &\mathit{choice}(\tau_1, \mathit{choice}(\tau_2, \tau_3)) \approx \mathit{choice}(\mathit{choice}(\tau_1, \tau_2), \tau_1) \\ &\mathit{product}(\tau_1, \mathit{product}(\tau_2, \tau_3)) \approx \mathit{product}(\mathit{product}(\tau_1, \tau_2), \tau_1) \\ &\mathit{product}(\tau_1, \mathit{choice}(\tau_2, \tau_3) \approx \mathit{choice}(\mathit{product}(\tau_1, \tau_2), \\ &\qquad \mathit{product}(\tau_1, \tau_3)). \end{aligned} \tag{3}$$

Note some of the implications of (3): a record which has **None** as one of its fields has no legal values. Similarly, a choice between **None** and any type τ type is τ since the **None** option can never be taken.

We will encode the polynomial types as formal multivariate polynomials with coefficients in $\mathbb{N}$. Each primitive type in Z is encoded as a distinct formal variable. The *choice* type construction operator is encoded as polynomial addition, while *product* is encoded as polynomial multiplication. **Unit** is encoded as 1. Thus, a pointer to X is encoded as $1 + X$, a pointer to a pointer to X is $2 + X$, etc. As might be expected, **None** is encoded as the polynomial 0.

Let $\mathbf{z}$ be a set of formal variables that are used for encoding Z. Then, the set of *polynomial types* is the set

$$\mathcal{P}_{\mathbf{z}} = \mathbb{N}[\mathbf{z}], \tag{4}$$

i.e., the set of all polynomials with natural coefficients over the set of formal variables $\mathbf{z}$. The subscript $\mathbf{z}$ is omitted when it is clear from context. It is mundane to see that $\mathcal{P}_{\mathbf{z}}$ has the same structure as the BNF (2), and that polynomial equality in it is isomorphic to the definition in (3).

We have that the set of all possible configurations is $[\mathbf{z}]$.

```
TYPE
Employee= Record
    id:Integer;
    name:String;
    isManager:Boolean;
    desc:Array [1..3] of Character;
    Case Integer of
        0:  yearly_salary:Real;
        1:  hourly_salary:Integer;
    end
end
```

Figure 5: An employee record in Pascal.

Example 4.1. Consider the Pascal[2] employee record defined in Figure 5. This type can be encoded as the polynomial

$$ISBC^3(R+I) \tag{5}$$

where I, S, B, C, and R are formal variables encoding the primitive types **Integer**, **String**, **Boolean**, **Character**, and **Real** (respectively).

As the example indicates, arrays of fixed size are considered syntactic sugar for product. This is justified since field names are ignored.

Henceforth, we will loosely refer to members of $\mathcal{P}$ both as polynomials and as types.

4.3 Type Equivalence in $\mathcal{P}$

We say that a polynomial is in an *expanded form* if it is written as sum of distinct monomials with coefficients. For algorithmic purposes, it is convenient to assume that the terms in the expanded form are sorted. Such a sort can be by any monomial ordering. The expanded form is canonical in the sense that the coefficient of each monomial is exactly the number of configurations of the sort of that monomial that the polynomial type generates.

Since type equality of the system (2), (3) is tantamount to polynomial equality, it has a straightforward implementation, provided the polynomials are given in an expanded form. However, as in Example 4.1, a type definition in a programming language does not directly yield this form. Expansion by applying the distributive rule and grouping monomials together may result in an exponential blowup of the size of the polynomial. The technique of *Zero equivalence testing* [45, Chap. 12] can be used to bring down the complexity of type equivalence in $\mathcal{P}$ to randomized polynomial time in the size of the *non-expanded* input.

All other algorithms presented here assume an expanded form of the input, and hence are potentially exponential in the input size.

4.4 Additive Subtyping in $\mathcal{P}$

How should subtyping of arithmetical types be defined? When we say that type T is a subtype of T' we mean that every value of T can be viewed also as a value of T'. When T and T' are thought of as sets of instances, then a simple way of interpreting this demand is that $T \subseteq T'$. Recall that each value of T contains not only an assignment of primitive values to a specific configuration, but also some sort of designation of the way that configuration was obtained in T. If this designation is specific to T, then it is unlikely

[2] We assume here a more modern version of Pascal which has a primitive **String** type.

that this value will also occur in T'. This is the reason why we have used a canonical enumeration of configurations for this designation. Canonical enumeration naturally gives rise to the following definition of subtyping of polynomial types.

Definition 4.2. For types $T, T' \in \mathcal{P}$, we say that T *is a subtype of* T' and write $T \sqsubseteq T'$ if $[m]T \leq [m]T'$ for all $m \in [\mathbf{z}]$.

Since Definition 4.2 means that for all types A and B, $A \sqsubseteq A + B$ and no other subtyping occurs, we call the $\sqsubseteq$ relation *additive* subtyping. The definition is also (trivially) equivalent to the following three subtyping rules for $\leq$ in the system defined by (2) and (3)

$$\tau_1 \leq \mathit{choice}(\tau_1, \tau_2)$$

$$\frac{\tau_1 \approx \tau_2}{\tau_1 \leq \tau_2} \qquad \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \tag{6}$$

Additive subtyping means that in going from a subtype to the supertype, more configurations can be added. In this process, the *structure* of any of the configurations of the subtype is not allowed to change. Thus, a record with 10 integer fields is a subtype of an array of 10 integers (these two types are in fact equivalent), which in turn is a subtype of a choice between an array of 10 integers and an array of 10 reals, which is a subtype of an array of 10 integers or reals. Also, two enumerated types in which all enumerated values are branded, stand in a subtype relationship if and only if their sets of values stand in a containment relationship. With pointers we have that X is a subtype of a pointer to X, which in turn is a subtype of a pointer to a pointer to X, etc.

When extended to recursive types, Definition 4.2 will mean that a list of integers is a subtype of a binary tree of integers. In general, additive subtyping captures very general and elaborate conversions, for example, the embedding of a binary tree in a forest of general trees.

According to Definition 4.2 each occurrence of a certain configuration of T can be matched against an occurrence of the same configuration of T', although as a result of the commutativity of addition, it is not specified how this matching is made. In the finite case, i.e., when the type system is limited to polynomial types the problems of *existence* of a configuration matching and *finding* a specific one are easy. Applying Definition 4.2 algorithmically, we can test for additive subtyping in linear time in the length of the expanded form of the input polynomials. (We are unaware of a more efficient procedure for this problem.) Definition 4.2 can also be used for *enumerating* all possible matchings between T and T'. It will be interesting to compare the problems again in the infinite case, i.e., when recursive types are allowed.

Note that if is it proved that the subtyping decision problem is undecidable, then it is clear that there is no algorithm which finds a configuration matching. Conversely, if $T \sqsubseteq T'$ was established, then it is possible to match the configurations in T with configurations in T' type in an orderly, deterministic fashion.

4.5 Multiplicative Subtyping in $\mathcal{P}$

Additive subtyping is a weak partial order on $\mathcal{P}$, which means that it is reflexive, transitive and anti-symmetric (excepting equality). Let us now define another weak partial order subtyping relationship on $\mathcal{P}$ which has these properties. In OO systems subtyping means that a type AB is a subtype of A for all types A and B. Let us write this as $AB \sqsubseteq^* A$.

The $\sqsubseteq^*$ relation is formalized in the following definition.

Definition 4.3. For types $T, T' \in \mathcal{P}$, we say that T *is a multiplicative subtype of* T' and write $T \sqsubseteq^* T'$ if there exists a type $T'' \in \mathcal{P}$ such that $T = T'T''$.

Clearly, the $\sqsubseteq^*$ relation is a weak partial order. In fact, it is not so difficult to see that it is equivalent to the relation $<^*$ defined in the system (2) and (3) by the following subtyping rules:

$$\mathit{product}(\tau_1, \tau_2) \leq^* \tau_1$$

$$\frac{\tau_1 \approx \tau_2}{\tau_1 \leq^* \tau_2} \qquad \frac{\tau_1 \leq^* \tau_2 \quad \tau_2 \leq^* \tau_3}{\tau_1 \leq^* \tau_3} \tag{7}$$

Determining whether two polynomial types stand in a multiplicative subtyping relationship is simply a matter of polynomial division—a problem for which efficient algorithms are known [13]. Again, efficiency is contingent on having the input in an expanded form.

Using our terminology, the value assignment to a configuration is truncated when going from a type to its multiplicative supertype. Thus, one can view multiplicative subtyping and Definition 4.2 as being complementary. Multiplicative subtyping deals with the values aspect of an instance, while additive subtyping deals with the configuration designation aspect.

4.6 Variations

The type system $\mathcal{P}$ with additive subtyping is the main type system we will deal with here. However, using similar mathematical machinery several other variations are possible.

1. The type system in which field ordering in a product is significant (product is not commutative) is represented by $\mathbb{N}[[\mathbf{z}]]$.
2. The type system that does not admit multiple occurrences of the same configuration is represented by $\mathbb{B}[\mathbf{z}]$. In such a system, the type $U_2 = \texttt{Int} + \texttt{Int}$ (Section 3) is equivalent to the primitive integer type.
3. In a similar fashion, $\mathbb{B}[[\mathbf{z}]]$ models a type system in which multiple occurrences of a configuration are considered one, but without commutativity of multiplication.

For each of these systems it is possible to apply the notion of additive subtyping, by using Definition 4.2 with the necessary changes. On the other hand, multiplicative subtyping as defined in Definition 4.3 only makes sense if multiplication is commutative.

The systems $\mathbb{N}[[\mathbf{z}]]$, $\mathbb{B}[\mathbf{z}]$ and $\mathbb{B}[[\mathbf{z}]]$ are discussed briefly in Section 9.

5 Intuition behind Algebraic Types

This section is devoted to an informal presentation of algebraic types, which are obtained by augmenting polynomial types with recursive definitions. The discourse will follow a series of examples highlighting some of their properties. Formal definitions are provided in the next section. A novel technique we present here is of solving recursive type equations using methods used in combinatorics for finding generating functions.

Example 5.1. Considering the C type definition in Figure 6, we see that the type `L` is defined using itself. Using the conventions introduced in the previous section, we may write this as an equation

$$L = I(1+L) = I + IL. \tag{8}$$

```
typedef struct L {
    int data;
    struct L *next;
};
```

Figure 6: A C definition of a linked list node data type.

One way of determining the meaning of the unknown type L from (8) is by directly solving (8) for L

$$L = \frac{I}{1-I}. \tag{9}$$

Clearly, this solution makes little sense in terms of types, since type division makes no sense at all, as does type subtraction. Nevertheless, writing the Taylor expansion about zero of the above we obtain

$$L = \sum_{i=1}^{\infty} I^i. \tag{10}$$

In words, (10) means that L is either one `int` or two `int`'s or three `int`'s, etc.

Yet another way of dealing with (8) is of repeatedly substituting L by its definition

$$\begin{aligned} L &= I + IL = I + I(I + IL) = I + I^2 + I^2L \\ &= I + I^2 + I^2(I + I^2 + I^2L) \\ &= I + I^2 + I^3 + I^4 + I^4L = \ldots \\ &= \sum_{i=1}^{\infty} I^i. \end{aligned} \tag{11}$$

Fortunately, both ways lead to the same infinite power series.

Example 5.2. Consider the binary search tree defined in Figure 2, which leads to the recursive definition equation

$$X = I(1+X)^2. \tag{12}$$

By moving terms we obtain

$$IX^2 + (2I-1)X + I = 0, \tag{13}$$

a quadratic equation with two solutions

$$X_{1,2} = \frac{1 - 2I \pm \sqrt{1-4I}}{2I}. \tag{14}$$

Carrying on while ignoring the senselessness of the extraction of the square root of types we see that the Taylor power series of X_1 contains terms with negative coefficients. Hence, this solution is meaningless for our purposes. The expansion of the second solution of (14) is more promising

$$X_2 = I + 2I^2 + 5I^3 + 14I^4 + \cdots. \tag{15}$$

In words, X_2 is either one `int` or one of two configurations of two `int`'s, or one of five configurations of three `int`'s, etc. Indeed, a binary tree has one configuration of one node, two configurations of two nodes, five configurations of three nodes etc.

It is a standard exercise in combinatorics to derive from (14) that

$$X_2 = \sum_{i=1}^{\infty} C_i I^i, \tag{16}$$

where C_n is the n^{th} Catalan number, defined by

$$C_n = \frac{1}{n+1}\binom{2n}{n}. \tag{17}$$

It is well known that C_n is the number of distinct binary trees with n nodes.

Again, repeated substitutions starting from (12) will eventually result in the same power series as (15):

$$\begin{aligned} X &= I(1 + 2X + X^2) \\ &= I(1 + 2I(1 + 2X + X^2) + I^2(1 + 2X + X^2)^2) \\ &= I + 2I + 4IX + 2I^2X^2 + I^3 + 2X^2I^2 + \cdots \end{aligned} \tag{18}$$

Let us introduce a notation for formal power series. The set of all power series, finite and infinite, of monomials $[\mathbf{x}]$ with coefficients in $\mathbb{A}$ is denoted by $\mathbb{A}\langle\mathbf{x}\rangle$. Since $0 \in \mathbb{A}$ we have $\mathbb{A}[\mathbf{x}] \subseteq \mathbb{A}\langle\mathbf{x}\rangle$. For completeness of the notation we let $\mathbb{A}\langle\langle\mathbf{x}\rangle\rangle$ denote the set of *pseudo formal power series*, i.e., the formal series with coefficients from $\mathbb{A}$ and pseudo monomials from $\mathbf{x}^*$. For a formal power series $p \in \mathbb{A}\langle\mathbf{x}\rangle$ we let $[m]p$ denote again the coefficient of monomial m in p. The same convention can be applied to pseudo formal power series.

Since the series are formal, we feel free to multiply and add them without concerning ourselves too much with questions of convergence.

Examples 5.1 and 5.2 indicate that it might be possible to encode recursive data types with power series from $\mathbb{N}\langle\mathbf{z}\rangle$. The power series of a type, also called *generating function* of the type, is uniquely defined by the demand that the coefficient of a certain monomial is the number of different ways the configuration associated with this monomial occurs in the type. We will make scant distinction between a recursive type and its power series.

We do not assume an extensive background in generating functions and their applications in combinatorics. (The curious reader may want to consult standard textbooks on the topic, e.g., [21, 39].) However, a few words are in place here to explain why solving directly for an unknown type yielded correct results, even though invalid operations were used along the way. The standard technique for discovering an explicit representation of a generating function is to search for an equation which this function must satisfy. After solving this equation, we look for the generating function among its, hopefully not too many, solutions.

We have essentially repeated this standard technique here, except that not much work was required to discover such an equation. The recursive type equation is an equation which the generating function of the type must satisfy. Therefore, the generating function of the type must be found among the solutions of the type equation.

There are cases in which a recursive type has no generating function since it has an infinite number of basic configurations.

Example 5.3. Consider the type D defined by Figure 7, which gives rise to the type equation

$$D = I + (1 + D). \tag{19}$$

This equation has no solutions since it is equivalent to $I = -1$.

Further, repeated substitutions fail to converge

$$\begin{aligned} D &= 1 + I + D = 1 + I + (1 + I + D) = \cdots \\ &= n(1 + I) + D \end{aligned} \tag{20}$$

```
typedef union D {
    int data;
    union D *next;
};
```

Figure 7: A degenerate type definition in C.

for all $n \geq 1$. Examining type D directly we see the same phenomenon. The type generates an infinite number of ways of getting a single integer, as well as an infinite number of ways of getting a configuration with no primitive data types at all.

Types which exhibit this sort of behavior are called *degenerate* types. Degenerate types are explicitly excluded from our attention. We argue also that such types make little sense from a programming perspective. If a type has a countably infinite number of a specific configuration $m \in [\mathbf{z}]$, then this infinity is better represented by a single configuration mI where the type I is an integer used for denoting the selection of the particular m.

Definition 5.4. A recursive arithmetical type is called degenerate if in the process of repeated substitutions a monomial with an unbounded coefficient is generated.

To our knowledge, this is the first time this family of degenerate types is identified. Even though from a practical point of view degenerate types should not be used, they have some value from a theoretical point of view. For example, it possible to define the type N of natural numbers as

$$N = 1 + N. \tag{21}$$

This technique cannot however be extended to define a type of real numbers since a cardinality $\aleph_1$ cannot be obtained using recursive definitions.

Generating functions are by large formal mathematical creatures. It is not necessary that they converge in order to manipulate them. We define a notion of substitution of a formal power series into a polynomial: For a polynomial P over a set of formal variables $\mathbf{x}$, a set $\mathbf{w}$ of formal power series, $\mathbf{w} \subseteq \mathbb{Z}\langle\mathbf{z}\rangle$, and a mapping of $\mathbf{x}$ to $\mathbf{w}$, the substitution of $\mathbf{x}$ in P by $\mathbf{w}$, $P[\mathbf{x}/\mathbf{w}]$ is defined in the natural way, i.e., by a substitution of each member of $\mathbf{x}$ by its map in $\mathbf{w}$. It can be shown [27] that this substitution process, although algorithmically impossible to carry out, is well defined, and that for all $\mathbf{w}$,

$$P[\mathbf{x}/\mathbf{w}] \in \mathbb{Z}\langle\mathbf{z}\rangle \tag{22}$$

Moreover, we have that substitutions of formal power series in polynomials obey the usual rules of polynomial substitution.

Fact 5.5. For all polynomials P_1, P_2 and all $\mathbf{w}$,

$$\begin{aligned} {}_1[\mathbf{x}/\mathbf{w}] + P_2[\mathbf{x}/\mathbf{w}] &= (P_1 + P_2)[\mathbf{x}/\mathbf{w}] \\ P_1[\mathbf{x}/\mathbf{w}]P_2[\mathbf{x}/\mathbf{w}] &= (P_1P_2)[\mathbf{x}/\mathbf{w}] \end{aligned} \tag{23}$$

Most of the literature on generating functions deals with univariate generating functions. We cannot accept such limitation here since recursive data types often use more than one primitive type. Our next example is a recursive type built upon two primitive types.

Example 5.6. Consider the recursive type defined by

$$B = 1 + z_1(1 + z_2)B \tag{24}$$

In words, B is a pointer to a record containing a z_1, a pointer to z_2 and another pointer of type B. It is possible to show from (24) that

$$[z_1^n z_2^m]B = \binom{n}{m}. \tag{25}$$

However, it is much easier to derive (25) from elementary considerations. Type B is simply a linked list of nodes such that each one of them contains a z_1 and an optional z_2. Suppose that the list has n nodes, i.e., n times z_1 data. Then, there are $\binom{n}{m}$ ways of selecting m nodes from this list of n nodes, on which m data items of primitive type z_2's can be placed.

Sometimes it is convenient to use two mutually recursive definitions to define a type. A simple example is that of the binary tree definition in Pascal (Figure 1). In this definition, type `T1` is defined using type `Y` and vice versa. Here is a slightly more interesting example.

Example 5.7. A red-black tree is a binary search tree in which node colors are used for balancing the tree [12, Chap. 14]. In such a tree every node is colored either red or black and if a node is red then both its children must be black. This gives rise to the following system of equations:

$$\begin{aligned} T_r &= 1 + zT_b^2 \\ T_b &= 1 + z(T_b + T_r)^2. \end{aligned} \tag{26}$$

The analytic solution of (26) is in terms of roots of a quartic polynomial equation and will not be written here.

Both in the case of the binary tree and the red-black tree, it is possible to use straightforward transformations and write the two type equations as one. However, in general, this is not always possible.

Example 5.8. Consider the following system of type equations

$$\begin{aligned} M_1 &= z(M_1 + M_0) \\ M_0 &= 1 + zM_0 \end{aligned} \tag{27}$$

Then, any attempt to eliminate M_0 from the system (27) to obtain a single type equation for M_1 will yield an invalid such equation, namely one in which there is a subtraction. We will devote more attention to types M_0 and M_1 below in Section 8.

6 Algebraic Types

In order to add recursive type definition to the BNF (2), we need an unbounded set of *type variables*: $\sigma_1, \sigma_2, \ldots$, and a *recursive type definition* operator which will be denoted by μ. The BNF (2) is then extended with

$$\begin{aligned} \tau &::= \sigma_1 \mid \sigma_2 \ldots \\ \tau &::= \mu\sigma_i.\tau \end{aligned} \tag{28}$$

and with the additional restriction that a type variable σ_i can only be used within a context of a $\mu\sigma_i$ expression.

A much cleaner representation of the algebraic types is given by the following definition.

Definition 6.1. The set $\mathcal{A}_s$ of algebraic types consists of all types T that can be defined by a system of polynomial equations:

$$\begin{aligned} T_1 &= P_1(T_1, \ldots, T_n, \mathbf{z}) \\ &\vdots \\ T_n &= P_n(T_1, \ldots, T_n, \mathbf{z}) \end{aligned} \tag{29}$$

where $P_1, \ldots, P_n \in \mathbb{N}[T_1, \ldots, T_n, \mathbf{z}]$ and $T \in \{T_1, \ldots, T_n\}$.

As before, the subscript $\mathbf{z}$ is omitted whenever it is clear from context.

Note that there are two kinds of players in (29):

Formal Variables These are the primitive types $\mathbf{z}$.

Unknowns These are the newly defined, mutually recursive types $T_1, \ldots, T_n$.

Eq. (29) defines the unknowns in term of the formal variables. We can say that the unknowns are a *function* of the formal variables. The term "algebraic" was coined for our recursive types since this function is algebraic in the algebraic geometry sense:

Definition 6.2. Let $\mathbf{x} = \langle x_1, \ldots, x_m \rangle$ and $\mathbf{y} = \langle y_1, \ldots, y_n \rangle$. An *algebraic function* is a multivariate multi-valued function $f : \mathbb{C}^m \to \mathbb{C}^n$ mapping $\mathbf{x}$ to $\mathbf{y}$, defined by a system of implicit polynomial equations

$$\begin{aligned} P_1(\mathbf{x}, \mathbf{y}) &= 0 \\ &\vdots \\ P_n(\mathbf{x}, \mathbf{y}) &= 0 \end{aligned} \tag{30}$$

where $P_i \in \mathbb{C}[\mathbf{x}, \mathbf{y}]$.

Clearly, (29) is an instance of (30). Therefore, each member of $\mathcal{A}$ is also what is called a *branch* of an algebraic function. On the other hand, there are algebraic functions which are not algebraic types. For example, the function

$$y = y(x) = \frac{1}{1+x} = \sum_{i=0}^{\infty} (-1)^i x^i \tag{31}$$

has only one branch which clearly does not correspond to a type.

There are at least three strategies for finding the generating function of a type from its definition. The first is by a process of repeated substitutions. Even though this process is infinite, in some cases it is possible to infer about it and deduce the infinite power series.

Such deduction is complicated by the fact that there are infinitely many different ways of carrying out the substitutions. At each substitution stage, one can choose any type T_i, any definition for it (the original one, i.e., P_i, or any alternative polynomial obtained from P_i through previous substitutions), and replace it into the current definition of any other unknown T_j. The only restriction is that each one of T_i is selected an infinite number of times.

A priori, it is not clear that these different ways will always result in the same power series. It is necessary to develop specialized mathematical machinery to deal with the notion of "convergence" into an infinite, multi-dimensional series. An excellent treatise of these topics is provided in Kuich and Salomaa's book [27], and will not be repeated here. It is however not too difficult to show that if

$$\begin{aligned} [1]P_i &\neq 0 \\ [z_j]P_i &= 0 \end{aligned} \tag{32}$$

for all $1 \leq i \leq n$ and all $1 \leq j \leq k$, then the types defined by (29) are not degenerate.

The second strategy for finding the infinite power series of a type is described in [27]: start from an initial approximation that T_i is 0 for all $1 \leq i \leq n$. Then, the $(\ell+1)^{th}$ approximation is obtained from the ℓ^{th} approximation by substituting it into the system (29). This strategy can deal with equations such as

$$T_1 = T_1^2, \tag{33}$$

and

$$T_2 = zT_2 + 2T_2 \tag{34}$$

which do not define degenerate types as defined in Definition 5.4, even though a repeated substitution process in them does not appear to "converge". Note that this strategy is similar to the familiar fixed point iteration method for finding the minimal solution (with respect to the additive subtyping partial order) in a system of equations, as used in more traditional type theory.

It is shown in [27] that if

$$\begin{aligned} [1]P_i &= 0 \\ [z_j]P_i &= 0 \end{aligned} \tag{35}$$

for all $1 \leq i \leq n$ and all $1 \leq j \leq k$ (a *proper* set of equations) or if

$$[m]P_i = 0 \tag{36}$$

for all $1 \leq i \leq n$ and all $m \in [\mathbf{z}] - 1$ (a *weakly strict* set of equations) then this successive approximation process converges to a solution.

The third strategy of deducing the power series is by solving the system (29) analytically, then writing the Taylor series of all solutions, and selecting the one which corresponds to the sought generating function. However, since there is no analytic solution to quintic and higher order equations, this is impossible to do in the general case.

The third strategy does not yield the same results as the second for the type defined by (33). There are two branches to the analytic solution of (33), $T_1 = 0$ and $T_1 = 1$. Indeed, both `Unit` and `None` satisfy (33). In (34) on the other hand, the second and third strategies agree, while the first fails.

There is an interesting family of types in which it is always possible to find an analytic solution of (29). Regrettably, even in this family it is not in general possible to write an explicit expression for the coefficients of the sought power series.

Definition 6.3. The set $\mathcal{R} \subset \mathcal{A}$ of *rational types* is defined by the following condition. Type $T \in \mathcal{R}$ if and only if it can be defined by a system of equations

$$\mathbf{T} = \mathbf{P}\mathbf{T} + \mathbf{Q} \tag{37}$$

where $\mathbf{T}$ is an $(n \times 1)$ vector of unknowns, $T \in \mathbf{T}$, while $\mathbf{P}$ is an $(n \times n)$ matrix, Q is an $(n \times 1)$ vector, and the elements of $\mathbf{P}$ and $\mathbf{Q}$ are in $\mathbb{N}[\mathbf{z}]$.

The system (37) is nothing but a system of linear equations in the unknown types. It is therefore possible to employ Gaussian elimination and compute the type T as an explicit function of the coefficients in this system. Since all coefficients in (37) are polynomials it follows that the generating function of T is given by

$$T = P(\mathbf{z})/Q(\mathbf{z}) \tag{38}$$

where $P, Q \in \mathbb{Z}[\mathbf{z}]$. Eq. (38) explains why this restricted family of algebraic types are called rational types.[3] From a programmer standpoint, rational types are recursive types which have the property that no recursive type definition makes more than a single use of each of the user defined types in each record.

It is easy to verify from Definition 6.3 that $\mathcal{R}$ is closed under additions and multiplications. This observation will be used in the following section.

[3] Rational types are not to be confused with rational trees, a mathematical device sometimes used in the study of recursive systems.

7 Additive Subtyping in $\mathcal{R}$ is Undecidable

In this section we focus our attention on additive subtyping (henceforth just subtyping). We will show that subtyping in $\mathcal{R}$ is undecidable. Of course, this implies that the more general problem of subtyping in $\mathcal{A}$ is undecidable as well. It is not clear however if the proof can be made any simpler by working in $\mathcal{A}$, or whether there are interesting properties of $\mathcal{A}$ which do not hold in $\mathcal{R}$.

The proof is carried out by reduction from Hilbert's tenth problem, the solution of polynomial Diophantine equations. We will see that for every such equation, there is a pair of rational types, which stand in a subtype relationship if and only if the equation is solvable.

Definition 7.1. *Hilbert's tenth problem (H10):*

Instance: A multivariate polynomial $Q(u_1, \ldots, u_k) \in \mathbb{Q}[u_1, \ldots, u_k]$.

Question: Is there an assignment of rational numbers to $u_1, \ldots, u_k$ such that

$$Q(u_1, \ldots, u_k) = 0. \tag{39}$$

David Hilbert presented H10 in his now famous 1900 lecture before the second *International Congress of Mathematicians*, as part of the set of 23 problems, which he deemed as the challenge left to the 20^{th} century mathematics by the 19^{th}. H10 was the only decision problem in this set. In fact, it is the only one which can be thought of as a computer science problem.

Surviving attacks by J. Robinson, M. Davis, H. Putnam and others, H10 finally yielded to Yuri Matiyasevich [29, 36][4] who provided the missing step in the proof that it is undecidable.

Fact 7.2. H10 is undecidable.

In this paper, we will be concerned only in the variant of H10 in which the coefficients of Q are restricted to be integers, i.e., $Q \in \mathbb{Z}[\mathbf{u}]$ and the sought solutions are restricted to be natural numbers instead of integers. The restriction does not lose any generality [30, Chap. 1], and the restricted form of H10 is undecidable as well.

The proof of Fact 7.2 uses what is called in the literature a universal Diophantine equation.

Definition 7.3. Let $U \in \mathbb{Z}[c_1, \ldots, c_l, u_1, \ldots, u_r]$ be a polynomial with integer coefficients in the *code parameters* $c_1, \ldots, c_l$ and the unknowns $u_1, \ldots, u_r$. Suppose that for every given equation of the form (39), there is a setting of the code parameters such that the equation

$$U(c_1, \ldots, c_l, u_1, \ldots, u_r) = 0, \tag{40}$$

is solvable exactly when the given equation is solvable. Then, Eq. (40) is called a *universal Diophantine equation.*

Note that each instance of (40) has r unknowns and that in general $r \neq k$. The existence of universal equations will allow us to carry out the reduction from a restricted set of instances of H10, rather than the whole range of those instances.

We will employ a standard technique of using a generating function to enumerate the values of a multivariate function of the natural numbers (such a function can also be thought of as a multi-dimensional series).

[4] Several distinct transliterations of this famous Russian mathematician were used in the literature

Definition 7.4. Given $S(u_1, \ldots, u_k)$, a multivariate function of natural arguments, its *enumerating generating function* (or, for short, *enumerating function*) is

$$F_S(\mathbf{z}) = \sum_{u_1=0}^{\infty} \cdots \sum_{u_k=0}^{\infty} S(u_1, \ldots, u_k) z_1^{u_1} \cdots z_k^{u_k} \tag{41}$$

If S assumes only natural values, then its enumerating function F_S is in $\mathbb{N}(\{z_1, \ldots, z_k\})$ and therefore may correspond to an algebraic type. We call this type the *enumerating type* of F.

The following lemma is pertinent to our reduction.

Lemma 7.5. For all $Q \in \mathbb{N}[\mathbf{u}]$, the function T_Q, the enumerating function of Q, belongs to $\mathcal{R}$.

Lemma 7.5 can be used as a black-box in the reduction. We therefore postpone the presentation of its proof to Section 8 below. The following example will help understand the lemma, as well as Definition 7.4.

Example 7.6. Consider the polynomial

$$Q(u_1, u_2) = u_1^2 u_2 + 1. \tag{42}$$

We can tabulate the values of Q on all natural assignments to u_1 and u_2 in an infinite two dimensional table as demonstrated in Table 1.

		u_1					
		0	1	2	3	4	$\cdots$
u_2	0	1	1	1	1	1	$\cdots$
	1	1	2	5	10	17	$\cdots$
	2	1	3	9	19	33	$\cdots$
	3	1	4	13	28	49	$\cdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Table 1: The values of $u_1^2 u_2 + 1$.

We can summarize the enumeration of this two dimensional table using a generating function $F(z_1, z_2)$. Reading the values along the secondary diagonals of Table 1 we can write the first few terms of F:

$$\begin{aligned} F(z_1, z_2) = 1 &+ (z_1 + z_2) + (z_1^2 + 2z_1 z_2 + z_2^2) \\ &+ (z_1^3 + 5z_1^2 z_2 + 3z_1 z_2^2 + z_2^3) + \\ &+ (z_1^4 + 10 z_1^3 z_2 + 9 z_1^2 z_2^2 + \cdots) + \cdots \end{aligned} \tag{43}$$

Notice that z_1 and z_2 are formal variables used in enumerating the values that Q assumes, and therefore play an entirely different role than that of u_1 and u_2. Variables z_1 and z_2 typically range over $\mathbb{C}$, while $u_1, u_2 \in \mathbb{N}$.

Examining (43) we see that the coefficients of F increase at a polynomial rate, and therefore, F is well defined in some neighborhood of the origin. Eq. (43) does not tell us much more about the nature and the behavior of the function F. However, using Lemma 7.5 we can assert that F is a rational function. Moreover, it is a rational type.

We are now ready to present the main result of this paper.

Theorem 7.7. *Subtyping in $\mathcal{R}_{\mathbf{z}}$ is undecidable if $|\mathbf{z}| \geq 9$.*

Proof. By reduction from H10. Given an arbitrary Diophantine equation

$$Q(u_1, \ldots, u_k) = 0. \tag{44}$$

we write the inequality

$$1 - (Q(u_1, \ldots, u_k))^2 > 0. \tag{45}$$

Since the coefficients of Q and $u_1, \ldots, u_k$ are all integers, the values that Q assumes are integers as well. It follows that the set of solutions of (44) is the same as that of (45).

By rearranging terms in (45) we can rewrite it as

$$Q_1(u_1, \ldots, u_k) > Q_2(u_1, \ldots, u_k), \tag{46}$$

where Q_1 and Q_2 are polynomials with *natural* coefficients.

Using Lemma 7.5 we now construct T_{Q_1} and T_{Q_2}, the enumerating types of Q_1 and Q_2. We argue that

$$T_{Q_1} \sqsubseteq T_{Q_2} \tag{47}$$

if and only if there is no solution to (44).

Suppose that (47) holds. Then, every coefficient of T_{Q_1} is no greater than the corresponding coefficient of T_{Q_2}, i.e.,

$$Q_1(u_1, \ldots, u_k) \leq Q_2(u_1, \ldots, u_k) \tag{48}$$

for any setting of $u_1, \ldots, u_k$. This implies that Ineq. (46) never holds, and therefore Eq. (44) is unsolvable. The opposite direction is carried out similarly.

Note that in this construction, k, the number of unknowns in (44) is exactly the same as the number of primitive types upon which types T_{Q_1} and T_{Q_2} are constructed. Fortunately, and thanks to the existence of universal equations, this does not mean that the type system is required to have an unbounded number of primitive types. We only need to carry out the reduction for all instances of a universal equation of the form (40).

Our proof is completed by noting that there exists a universal equation with nine unknowns. (Such an equation is described in [24]). □

8 Proof of Lemma 7.5

In this section we will see how all polynomials can be enumerated by types in $\mathcal{R}$, whereby proving Lemma 7.5, and completing the proof that subtyping in $\mathcal{R}$ is undecidable. The most difficult step is in showing that all univariate monomials have an enumerating rational type. After doing so, we extend this claim to multivariate monomials and subsequently to multivariate polynomials.

Let us use the convention that $0^0 = 1$.

Lemma 8.1. For all $r \geq 0$, there exists a rational type M_r such that

$$M_r = \sum_{i=0}^{\infty} r^i z^r, \tag{49}$$

Thus,

$$\begin{aligned} M_0 &= 1 + z + z^2 + z^3 + \cdots \\ M_1 &= z + 2z^2 + 3z^3 + \cdots \\ M_2 &= z + 4z^2 + 9z^3 + \cdots \\ M_3 &= z + 8z^2 + 27z^3 + \cdots \\ &\vdots \end{aligned}$$

A few words of intuition are in place before we proceed to the proof. Type M_0 is the linked list of z's, which can be defined by

$$M_0 = 1 + zM_0 \tag{50}$$

Clearly, M_0 is a rational type and

$$M_0 = \sum_{i=0}^{\infty} z^i. \tag{51}$$

Using M_0 we can define M_1 by

$$M_1 = zM_1 + zM_0. \tag{52}$$

By unfolding we obtain

$$\begin{aligned} M_1 &= z(zM_1 + zM_0) + z + z^2 + \cdots \\ &= z(zM_1 + z + z^2 + \cdots) + z + z^2 + \cdots \\ &= z^2 M_1 + z + 2z^2 + 2z^3 + \cdots \\ &= z^2 M_1 + z + 2z^2 + 2\sum_{i=3}^{\infty} z^i \\ &= z + 2z^2 + z^2 M_1 + 2z^3 M_0. \end{aligned} \tag{53}$$

Further unfolding of the terms $z^2 M_1$ and $2z^3 M_0$ will yield only z^3 and higher order terms. Therefore, the first two terms in the expansion of M_1 must be $z + 2z^2$. This argument establishes an induction base for a proof by induction that

$$[z^n] M_1 = n \tag{54}$$

for all $n \geq 0$.

To determine $[z^n] M_1$ for $n \geq 0$, examine the right hand side of Eq. (52). By the inductive hypothesis we have that for the term zM_1 $[z^n] zM_1 = n - 1$. On the other hand, $[z^n] M_0 = 1$. Thus $[z^n] M_1 = (n-1) + 1 = n$.

In a similar fashion, we can define the type M_2 as

$$M_2 = zM_2 + 2zM_1 + zM_0. \tag{55}$$

The induction step in this case can be carried out as follows. By the inductive hypothesis

$$[z^n] zM_2 = (n-1)^2. \tag{56}$$

From (54) and (51) we have

$$[z^n] 2zM_1 = 2(n-1) \tag{57}$$

and

$$[z^n] zM_0 = 1. \tag{58}$$

Summing (56), (57) and (58), we obtain $[z^n] M_2 = (n-1)^2 + 2n + 1 = n^2$. The proof of Lemma 8.1 is a generalization of these considerations.

Proof of Lemma 8.1.

Let the types M_r, $r \geq 0$ be defined by (50) and the system of mutually recursive equations:

$$M_r = z \sum_{i=0}^{r} \binom{r}{i} M_{r-i} \tag{59}$$

Then, by means of simultaneous induction on n and r, we show that

$$[z^n] M_r = n^r \tag{60}$$

for all $n \geq 0, r \geq 0$.

The base case $r = 0, n \geq 0$ trivially follows from the expansion of the linked list.

Consider the case $n = 0, r \geq 0$. It is clear from the definition (59) that $[z^0]M_r = 0$ for all $r = 1, 2, \ldots$. In other words, only M_0 has a constant term in its expansion.

The induction step is the case $n > 0, r > 0$. We use (59) and then the inductive hypothesis to obtain

$$\begin{aligned}
[z^n]M_r &= [z^n]\sum_{i=0}^{r}\binom{r}{i}M_{r-i} \\
&= \sum_{i=0}^{r}\binom{r}{i}[z^n]M_{r-i} \\
&= \sum_{i=0}^{r}\binom{r}{i}(n-1)^{r-i} \\
&= \big((n-1)+1\big)^r \\
&= n^r
\end{aligned}$$

In order to show that all monomials can be enumerated by rational types we need the following lemma which shows how to enumerate a product.

Lemma 8.2. Let $S_1(u_1, \ldots, u_l)$ and $S_2(u_{l+1}, \ldots, u_k)$ be two multivariate functions of the naturals whose sets of formal arguments are disjoint. Let S be their product: $S(u_1, \ldots, u_k) = S_1(u_1, \ldots, u_l)S_2(u_{l+1}, \ldots, u_k)$. Then, F_S, the enumerating function of S, is

$$F_S(z_1, \ldots, z_k) = F_{S_1}(z_1, \ldots, z_l)F_{S_2}(z_{l+1}, \ldots, z_k)$$

where F_{S_1} and F_{S_2} are the enumerating functions of S_1 and S_2.

Proof. Considering Definition 7.4, we see that in multiplying two sums of the form (41), where all summation indices are disjoint, gives another sum of the form (41). □

Combining lemmas 8.1 and Lemma 8.2, we obtain that an enumeration of all multivariate monomials.

Corollary 8.3. All $m \in [\mathbf{z}]$ can be enumerated by a rational type.

The missing step for the enumeration of all polynomials is the enumeration of addition:

Lemma 8.4. Let $S_1(u_1, \ldots, u_k)$ and $S_2(u_1, \ldots, u_k)$ be two multivariate functions of the naturals, and let S be their sum: $S(u_1, \ldots, u_k) = S_1(u_1, \ldots, u_k) + S_2(u_1, \ldots, u_k)$. Then, F_S, the enumerating function of S, is $F_S(z_1, \ldots, z_k) = F_{S_1}(z_1, \ldots, z_k) + F_{S_2}(z_1, \ldots, z_k)$ where F_{S_1} and F_{S_2} are the enumerating functions of S_1 and S_2.

Proof. The proof follows immediately from Definition 7.4. □

The proof of Lemma 7.5 now follows from Lemma 8.4 and Corollary 8.3.

In summary, we have shown that the enumerating function of polynomials in $\mathbb{N}[\mathbf{u}]$ is a rational type. It is not difficult to extend our proof to show that the encoding function of polynomials in $\mathbb{Z}[\mathbf{u}]$ is a rational *function*. Conversely, it follows from Matiyasevich's proof that all computable functions can be expressed as the values of polynomials in $\mathbb{Z}[\mathbf{u}]$. Since the coefficients of rational (and algebraic) types can clearly be computed, then, for any such type, there is a polynomial in $\mathbb{Z}[\mathbf{u}]$ whose values span these coefficients.

Obviously, rational types cannot enumerate all polynomials in $\mathbb{Z}[\mathbf{u}]$, since such polynomials may assume negative values. The converse, namely whether polynomials in $\mathbb{N}[\mathbf{u}]$ are sufficient for the enumeration of rational types is not clear.

9 Discussion and Open Problems

Perhaps the most interesting definitional problem that this paper leaves open is that of a notion of "structured conversion" between arithmetical types, beyond, or on top of, the abstract CCG mechanism. Such a notion may lead to a different definition of additive subtyping which might be decidable.

As an indication that structured convertibility might be easier than mere inclusion should serve the fact that containment of context free languages is undecidable, while there is an algorithm (albeit complicated) for the structured version of this problem, namely containment of parenthesized grammars [37, Chap. VIII.3].

Other, more mathematically oriented problems and directions for further research are mentioned below.

9.1 The main result, extensions and improvements

The main result presented in this paper is that additive subtyping is undecidable in $\mathbb{N}\langle\mathbf{z}\rangle$. Our undecidability result required that the type system has 9 or more primitive types. The value 9 seems to be borderline. Java for example has 8 primitive types, while C (depending on the counting) has more than 9. Concrete subtyping problems may use a fewer number of primitive types than those which exist in the host programming language. Therefore it is interesting to try to reduce the number of primitive types required for the proof.

Reducing the number of unknowns in a universal polynomial Diophantine seems difficult. The lower bound of 9 achieved by Jones [24] was not improved for almost twenty years. We believe that such a reduction is easier in our context. Support for this belief we find in the fact that there exists a universal *exponential* Diophantine equation using only three unknowns [30, Chap. 7]. We observe that the expressive power of rational types is richer than polynomials, and includes e.g., exponentiation. The type

$$E_r = 1 + rzE_r$$

gives an encoding of the function r^u for any integer $r > 0$. Using general algebraic types it is possible to encode even more interesting integer functions, such as the binomial values (Example 5.6), and in particular Catalan numbers (Example 5.2). Unfortunately, unlike polynomials, the composition of such integer functions is not so simple. Even a simple encoding in types of a function with doubly exponential growth rate does not seem possible. This makes the encoding the universal exponential equation with three unknowns using algebraic types an interesting challenge.

The following fact states that without commutativity, additive subtyping is undecidable with two primitive types or more.

Fact 9.1. Additive subtyping in $\mathbb{N}\langle\langle\mathbf{z}\rangle\rangle$, $|\mathbf{z}| \geq 2$ is undecidable.

Proof. The proof is by giving a type theoretical interpretation to the corresponding result in formal power series [38]. □

Fact 9.1 might be interpreted to gives another indication that a reduction in the number of unknowns is possible.

When there is only one primitive types, commutativity plays no role and we have that $\mathbb{A}\langle\langle z\rangle\rangle = \mathbb{A}\langle z\rangle$ for any ring $\mathbb{A}$. The most interesting case is $\mathbb{A} = \mathbb{N}$. It is a "celebrated" [5] open problem to show that additive subtyping in $\mathbb{N}\langle\langle\mathbf{z}\rangle\rangle$, $|\mathbf{z}| = 1$ is undecidable, or find an algorithm for it.

If choice is idempotent, specifically when $\mathbb{A} = \mathbb{B}$, then additive subtyping, and hence additive inequality can be decided in polynomial time.

[5] A. Salomaa (private communication), and P. Flajolet (private communication)

Fact 9.2. Additive subtyping in $\mathbb{B}\langle \mathbf{z} \rangle$ and in $\mathbb{B}\langle\langle \mathbf{z} \rangle\rangle$ can be decided in polynomial time.

Proof. The proof is by using Parikh's theorem [32] to show the infinite set, with duplications removed, of configurations generated by an algebraic type is a regular set. Details can be found in [27, Chap. 7]. □

Another research direction is to explore multiplicative subtyping in $\mathbb{B}\langle \mathbf{z} \rangle$, in $\mathbb{B}\langle\langle \mathbf{z} \rangle\rangle$, and $\mathbb{N}\langle\langle \mathbf{z} \rangle\rangle$

9.2 Other problems in the $\mathcal{A}$ type system

Given two systems of equations

$$\begin{aligned} T_1 &= P_1(T_1, \ldots, T_n, \mathbf{z}) \\ &\vdots \\ T_n &= P_n(T_1, \ldots, T_n, \mathbf{z}) \end{aligned} \tag{61}$$

and

$$\begin{aligned} T'_1 &= P'_1(T'_1, \ldots, T'_{n'}, \mathbf{z}) \\ &\vdots \\ T'_{n'} &= P'_n(T'_1, \ldots, T'_{n'}, \mathbf{z}), \end{aligned} \tag{62}$$

the type equality problem is to determine whether the Laurent power series expansion of the algebraic functions $T_1 = T_1(\mathbf{z})$ and $T'_1 = T'_1(\mathbf{z})$ are identical. A simple algorithm for this problem is by applying Tarski's theorem [42] which gives a quantifiers removal procedure and a checking algorithm for every first order predicate over the reals involving equality, addition and multiplication as well as the usual logical operators. If two algebraic functions coincide in a non-empty neighborhood of the origin, then their Laurent expansion must be identical. This condition is readily written in a prenex form, and hence can be tested.

Tarski's original algorithm is highly inefficient [15]. More efficient (though still non-polynomial) implementation of his theorem exist [10, 8, 11]. These algorithms use techniques related to Groebner bases [13]. A natural question is therefore whether Groebner bases can be applied directly for checking type equivalence. More generally, find a procedure for determining multiplicative subtyping in $\mathcal{A}$ (or show that it undecidable).

9.3 Function types

How should functions (in the programming sense) be incorporated into the arithmetical types framework? The natural way of doing so is using exponentiation: a function mapping type z_1 into type z_2 will be encoded as $z_2^{z_1}$. Exponential encoding models well currying and other common operations on function types. We may use the term *transcendental* for a type system which includes addition, product and exponentiation. Our undecidability result trivially extends to transcendental recursive type systems, and hence applies to traditional OO systems which admit functions as another type operator.

Exploring subtype and type equivalence issues in a non-recursive transcendental type system is interesting, but the parallels between the type system and the arithmetical encoding do not work as nicely: The encoding of a procedure type $\tau : z \to \texttt{Unit}$ as 1^z may lead to the false conclusion that $\tau = 1$. Also, subtyping of function types obeys the contra-variance rule for arguments, which would be hard to model using arithmetic.

9.4 A Restricted Type System

In the context of many contemporary OO languages, it is important to investigate a type system variant in which the choice type operator is used solely for the representation of references. Such a type system will not include types such as $z_1 + z_2$. However, it will include types such as $1 + z_1$ (a pointer to z), $2 + z$ (a pointer to a pointer to z) as well as $(1 + z_1)(2 + z_2)z_3$. This restriction roughly corresponds to languages such as Java and Eiffel in which there are no choices, but references which can be, to use the Eiffel terminology, ***void***. More formally, we define the set $\widetilde{\mathbb{N}}[\mathbf{x}] \subset \mathbb{N}[\mathbf{x}]$ as follows.

Definition 9.3. Given a finite set of primitive formal variables $\mathbf{x}$, the set $\widetilde{\mathbb{N}}[\mathbf{x}]$ of *pointer polynomials* over x is defined by

1. $x \in \widetilde{\mathbb{N}}[\mathbf{x}]$, for all $x \in \mathbf{x}$,
2. $1 + p \in \widetilde{\mathbb{N}}[\mathbf{x}]$ for all $p \in \widetilde{\mathbb{N}}[\mathbf{x}]$,
3. $p_1 p_2 \in \widetilde{\mathbb{N}}[\mathbf{x}]$ for all $p_1, p_2 \in \widetilde{\mathbb{N}}[\mathbf{x}]$, and
4. nothing else is in $\widetilde{\mathbb{N}}[\mathbf{x}]$.

The system $\widetilde{\mathbb{N}}[\mathbf{x}]$ is somewhat weird since it is closed under multiplication and substitution but not addition. An open problem is to determine the complexity of subtyping of algebraic types if the type equations are restricted to polynomials drawn from $\widetilde{\mathbb{N}}[\mathbf{x}]$.

Acknowledgments The comments of Jens Palsberg on a preliminary version of this paper are gratefully acknowledged.

References

[1] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, 1996.

[3] J. Auerbach. The μtype system. Unpublished Manuscript, IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, 1998.

[4] J. Auerbach, C. Barton, M. Chu-Carroll, and M. Raghavachari. Mockingbird: Flexible stub compilation from pairs of declarations. In M. G. Gouda, editor, *The 19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Austin, Texas, May-June 1999.

[5] J. Auerbach and M. Chu-Carroll. The Mockingbird system: A compiler based approach to maximally interoperable distributed programming. Technical Report RC20718, IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, 1997.

[6] J. Avotins, G. Maughan, and C. Mingins. Language processor construction: The case for YOOCC and TROOPER. In *Proceedings of TOOLS USA'95*, 1995.

[7] J. Avotins, C. Mingins, and H. Schmidt. Yes! an object-oriented compiler compiler YOOCC. In *Proceedings of TOOLS USA'95*, 1995.

[8] S. Basu and M.-F. Roy. On the combinatorial and algebraic complexity of quantifier elimination. *J. ACM*, 43(6):1002–1045, Nov. 1996.

[9] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33(4):309–338, May 1998.

[10] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proceeding of the 2nd GI Conference on Automata Theory and Formal Languages*, pages 134–183, New York, 1975. Springer Verlag.

[11] G. E. Collins. Quantifier elimination by cylindrical algebraic decomposition—twenty years of progress. In B. F. Caviness and J. R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 8–23. Springer Verlag, New York, 1998.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[13] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms*. Undergraduate Texts in Mathematics. Springer Verlag, second edition, 1996.

[14] F. Damm. Subtyping with union types, intersection types and recursive types. In Hagiya and Mitchell [20], pages 687–706.

[15] J. Davenport and J. Heintz. Real quantifier elimination if doubly exponential. *J. Symb. Comput.*, 5:29–35, 1988.

[16] R. Di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Birkhäuser, 1995.

[17] J. Gil and D. H. Lorenz. SOOP – A synthesizer of an object-oriented parser. In *Proceedings of the 16th International Conference on Technology of Object-Oriented Languages and Systems*, pages 81–96, Versailles, France, Mar. 6-10 1995. TOOLS 16 Europe Conference, Prentice-Hall.

[18] C. F. Goldfarb. *The SGML Handbook*. Clarendon Press, Oxford, 1990.

[19] C. F. Goldfarb and P. Prescod. *The XML Handbook*. Charles F. Goldfarb Series. Prentice-Hall, 1998.

[20] M. Hagiya and J. C. Mitchell, editors. *Proceedings of the 2nd International Symposium on Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, Sendai, Japan, Apr. 1994. Springer Verlag.

[21] M. Hofri. *Probabilistic Analysis of Algorithms*. Springer-Verlag New York Inc., 1987.

[22] J. Ichibia, editor. *Ada Programming Language*. ANSI/MIL-STD-1815A. Ada Joint Program Office, Department of Defense, Washington, DC, 1983.

[23] C. B. Jay. The FISh programming definition. Available as `http://www-staff.mcs.uts.edu.au/au/~cbj/-Publications/lastest_fish.ps.gz`, Oct. 1998.

[24] J. P. Jones. Universal Diophantine equation. *Journal of Symbolic Logic*, 47(3):547–571, 1982.

[25] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Software Series. Prentice-Hall, second edition, 1988.

[26] D. Kozen, J. Palsberg, and M. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5, 1995.

[27] W. Kuich and A. Salomaa. *Semirings, Automata, Languages*, volume 5 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1986.

[28] L. Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, 1986.

[29] J. V. Matijasevič. Enumerable sets are Diophantine. *Soviet Mathematics. Doklady*, 11(2):354–358, 1970. This is an English translation of the original paper in Russian (1970).

[30] Y. V. Matiyasevich. *Hilbert's tenth problem*. MIT Press, 1993.

[31] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[32] R. J. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, 1966.

[33] O. Patashnik. *BibTeXing*, 8 Feb. 1988. Documentation for general BibTeX users.

[34] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1991.

[35] M. Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1:71–89, 1991.

[36] J. Robinson. Hilbert's tenth problem. In *Proc. Symp. Pure Math.*, 20, pages 191–194. Amer. Math. Soc., Providence, Rhode Island, 1971.

[37] A. Salomaa. *Formal Languages*. Academic Press, New York and London, 1973.

[38] A. Salomaa and M. Soittola. *Automata-Theoretic Aspects of Formal Power Series*. Springer-Verlag, 1978.

[39] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley Publishing Company, Inc., 1996.

[40] T. Sekiguchi and A. Yonezawa. A complete type inference system for subtyped recursive types. In Hagiya and Mitchell [20], pages 667–685.

[41] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[42] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley, CA, second edition, 1951.

[43] N. Wirth. The programming language Pascal. *Acta Informatica*, 1:35–63, 1971.

[44] Recommendation x.208: Speficiation of abstract syntax notation one (ASN.1), Mar. 1988.

[45] R. E. Zippel. *Effective Polynomial Computation*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1993.